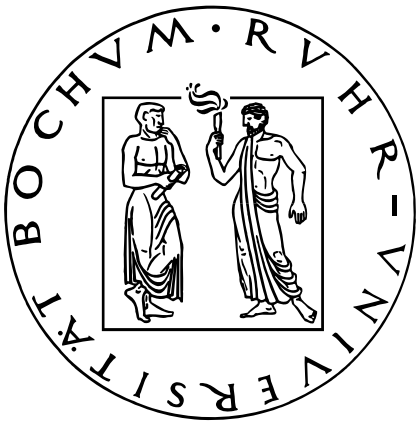
Elliptic Curve Cryptography as a Case Study for Hardware/Software Codesign

Lars Pontow

13.05.2004

Ruhr-Universität Bochum



Chair for Communication Security
Prof. Dr.-Ing. Christof Paar

Erklärung

Hiermit versichere ich, dass ich meine Diplomarbeit selbst verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Ort, Datum

Unterschrift

Abstract

Embedded systems, like Personal Digital Assistants (PDA) and mobile phones, are ubiquitous nowadays. With newer applications, like e-commerce, securing the vulnerable communication in these systems has become extremely important. For accomplishing this kind of security, asymmetric cryptography is required. But a major challenge when implementing asymmetric cryptographic algorithms on embedded systems is the limited CPU power and memory size. Hence dedicated hardware support to accelerate these algorithms is highly desirable. FPGAs are an attractive platform to implement such dedicated hardware in an inexpensive and uncomplicated way.

In this thesis, we analyze performance gain versus the hardware cost for elliptic and hyperelliptic curve cryptosystems, when a certain amount of special hardware is added to the system. For our implementation, we use a typical embedded processor, the ARM 7TDMI. Directly connected to the ARM processor is a XILNX VirtexE XCV2000E FPGA on which the special dedicated hardware is implemented. We implement ECC over $\mathbb{F}_{2^{167}}$ and HECC of genus 2 over $\mathbb{F}_{2^{81}}$. Thus, HECC provides about the same level of security as the ECC.

Our fastest ECC scalar multiplication is 1.9 ms at 25 MHz, which is 390.4 times faster than our implementation without dedicated hardware. We use 3220 slices on the FPGA for the dedicated hardware. The fastest HECC scalar multiplication takes 6.2 ms at 25 MHz using 1794 slices for the dedicated hardware, which is 248.4 times faster than the non-accelerated version.

Contents

List of Tables

List of Figures

1 Introduction

1.1 Motivation

Communication becomes an increasingly important part of our daily lives. With ever new advances in technology, like the ubiquitous internet or wireless communication, the amount of communication is rising steadily. The progress in communication however has also made the vulnerability of communication become more obvious. The necessity for securing communication has given a boost to the scientific field of cryptography over the past decades. The main objectives of cryptography comprise authenticity, integrity, and non-repudation of communication.

Today, two types of cryptographic algorithms exist: symmetric and asymmetric algorithms. Symmetric algorithms are the classical approach for secure communication. But they have the big drawback that communication partners have to share the same secret key, thus, the problem of key distribution arises.

The asymmetric principle was first publicly introduced in 1976 by Diffie and Hellman. It solves the key distribution problem in an elegant way by using two different keys, a public and a private one. Encryption and verifying involves a public key that is available to everyone. The private key is used for decryption and signing. However, asymmetric algorithms are computational much more expensive than symmetric algorithms, which is

viewed by many as the big knockout criteria for applications such as embedded systems with limited processor power and small sized memory.

In 1985, Miller [43] and Koblitz [28] independently proposed a public-key cryptosystem based on elliptic curves (EC). EC cryptosystems allow for shorter operands the same level of security than other cryptosystems, such as RSA or Diffie-Hellmann. This reduction in operand size is especially helpful in constrained environments like embedded systems. The idea that Jacobian groups of hyperelliptic curves (HEC) are suitable for cryptographical use was first introduced 1988 by Neal Koblitz [29]. Hyperelliptic curves are a special class of algebraic curves and can be viewed as generalization of elliptic curves. There are hyperelliptic curves of every genus $g \geq 1$. A hyperelliptic curve of genus $g = 1$ is an elliptic curve. The use of HECC allows to further reduce the operand size compared to ECC.

Due to the computational cost of public-key cryptosystems, dedicated hardware support is desirable. Reconfigurable hardware, like FPGAs, allows us to test different hardware solutions in an uncomplicated and inexpensive way, thus, avoiding the drawbacks of ASIC systems.

This thesis presents a tradeoff analysis of the cost in hardware and performance when implementing software functionality to a certain degree in special hardware. For this analysis we chose to implement an elliptic curve cryptosystem (ECC) over $\mathbb{F}_{2^{167}}$ and a hyperelliptic curve cryptosystem (HECC) of genus 2 over $\mathbb{F}_{2^{81}}$ on an embedded platform. The HECC over $\mathbb{F}_{2^{81}}$ implements about the same level of security as the ECC over $\mathbb{F}_{2^{167}}$. As target platform we use a combination of an ARM 7TDMI processor and reconfigurable logic.

The ARM 7TDMI is a typical processor for embedded systems. It is widely used for lots of different embedded applications. The FPGA used is a Xilinx VirtexE XCV2000E,

which provides enough hardware resources to implement complex architectures on it.

1.2 Thesis outline

Chapter 2 briefly introduces the mathematical background of EC and HEC cryptosystems. The emphasis lies on the mathematical basics relevant for this thesis.
Chapter 3 presents previous work done on ECC and HECC. The work presented lists different software and hardware realizations of such cryptosystems.
In Chapter 4 we describe the tools used and our target platform in more detail.
Chapter 5 covers our implementation of $\mathbb{F}_{2^m}$ field arithmetic. The first part of the chapter shows how we implemented the field arithmetic functions on the ARM 7TDMI. The second part of the chapter presents our implementations of the field arithmetic in special hardware on the FPGA.
In Chapter 6 we present the different design options that we implemented in this thesis together with the considerations that lead us to these options. The design options we introduce range from a software only option, running entirely on the ARM 7TDMI without any special hardware, up to a design option that can execute a complete scalar multiplication on the FPGA. We also present design options in between these two cases.
In Chapter 7 we finally present the timings we obtained for different design options. The timings we present are not just estimations, they are the results of real measurements of the implemented design options on our target platform. In this chapter we also discuss the results in terms of hardware cost and performance. We also present our conclusions here.

2 Mathematical Background

This chapter provides a short introduction to the mathematical background of Elliptic and Hyperelliptic Curve Cryptosystems. We only give a brief introduction covering all of the aspects that are relevant for this thesis. The parts 2.1 and 2.2 are mainly taken from [56]. The HECC part is taken from [71]. For a more detailed introduction to ECC we refer to the following literature [4,11,63,64]. For HECC we refer to [29–31,42].

2.1 Introduction to Finite Fields

A *finite field* consists of a finite set of elements F, two binary operations, addition and multiplication, and the additive and multiplicative inverses of each element. The binary operations satisfy certain arithmetic properties. The number of elements in the field is called the *order* of the finite field. There exists a finite field of order q if and only if q is a prime power. Essentially, there is only one finite field of order q denoted by $\mathbb{F}_q$. If $q = p^m$ where p is a prime and m is a positive integer, then p is called the *characteristic* of $\mathbb{F}_q$ and m is called the *extension degree* of $\mathbb{F}_q$.

In the following, we shortly describe the two most important types of finite fields applied in practice, the prime field $\mathbb{F}_p$ and the binary field $\mathbb{F}_{2^m}$.

2.1.1 The Finite Field $\mathbb{F}_p$

We call the finite field $\mathbb{F}_p$ where p is a prime number *prime field*. It is represented by the set of integers $\{0, 1, 2, \ldots, p-1\}$. The addition operation is *addition modulo* p, which means that for $a, b \in \mathbb{F}_p$, $a + b = r$, where r is the remainder of $a + b$ divided by p. The multiplication operation is *multiplication modulo* p, which means that for $a, b \in \mathbb{F}_p$, $a \cdot b = s$, where s is the remainder of $a \cdot b$ divided by p. If a is a non-zero element in $\mathbb{F}_p$, we say that the *inverse* of a modulo p, denoted by a^{-1}, is the unique integer $c \in \mathbb{F}_q$ for which $a \cdot c = 1$. In this thesis we do not use the finite field $\mathbb{F}_p$. We only use the finite field $\mathbb{F}_{2^m}$, which is presented in the following.

2.1.2 The Finite Field $\mathbb{F}_{2^m}$

The finite field $\mathbb{F}_{2^m}$ can be viewed as a vector space of dimension m over the field $\mathbb{F}_2$ which consists of the two elements 0 and 1. $\mathbb{F}_{2^m}$ is often referred to as *characteristic two finite field* or *binary finite field*. As it is a vector space, every element a of $\mathbb{F}_{2^m}$ can be represented as a bit string $(a_0 a_1 \ldots a_{m-1})$:

$$a = a_0 \cdot \beta_0 + a_1 \cdot \beta_1 + \cdots + a_{m-1} \cdot \beta_{m-1}, \text{where } a_i \in 0, 1.$$

The set $\{\beta_0, \beta_1, \ldots, \beta_{m-1}\}$ is called a *basis* of $\mathbb{F}_{2^m}$ over $\mathbb{F}_2$. There are many different bases and some of them lead to more efficient implementations than others. In this thesis, we only consider *polynomial basis representations*, because they are well suited to microprocessor and hardware architectures. Other bases are described, for example, in [26], which is also our main reference for this section. An irreducible polynomial of degree m over $\mathbb{F}_2$ can be written as $f(x) = x^m + f_{m-1}x^{m-1} + \cdots + f_2 x^2 + f_1 x + f_0$, where $f_i \in \{0, 1\}$. Irreducible means that it cannot be factored as a product of two

polynomials over $\mathbb{F}_2$, each of degree less than m. These so-called *reduction polynomials* $f(x)$ define a polynomial basis representation of $\mathbb{F}_{2^m}$, i.e.

$$\begin{aligned}\mathbb{F}_{2^m} &\simeq \{a_{m-1}x^{m-1} + \cdots + a_1x + a_0 : a_i \in \{0,1\}\} \\ &\simeq \{(a_{m-1} \ldots a_1a_0) : a_i \in \{0,1\}\}\end{aligned}$$

Thus, the elements of $\mathbb{F}_{2^m}$ can be represented by the set of all binary strings of length m. The multiplicative identity element is represented by the bit string $(00\ldots01)$ and the additive identity element is represented by the bit string of all 0's.
Addition is performed as bitwise XOR of the vector coefficients, i.e. if we have two elements of $\mathbb{F}_{2^m}$, $a = (a_{m-1} \ldots a_1a_0)$ and $b = (b_{m-1} \ldots b_1b_0)$, then $a+b = c = (c_{m-1} \ldots c_1c_0)$, where $c_i = a_i + b_i \mod 2$.
If $a = (a_{m-1} \ldots a_1a_0)$ and $b = (b_{m-1} \ldots b_1b_0)$ are elements of $\mathbb{F}_{2^m}$ multiplication is performed as follows: $a \cdot b = r = (r_{m-1} \ldots r_1r_0)$, where the polynomial $r(x) = r_{m-1}x^{m-1} + \cdots + r_1x + r_0$ is the remainder when the polynomial

$$(a_{m-1}x^{m-1} + \cdots + a_1x + a_0) \cdot (b_{m-1}x^{m-1} + \cdots + b_1x + b_0)$$

is divided by the reduction polynomial $f(x)$.

2.2 Introduction to Elliptic Curves

In 1985, Miller [43] and Koblitz [28], independently proposed a public-key cryptosystem analogous to the ElGamal schemes [13] in which the multiplicative group of integers modulo p, denoted by $\mathbb{Z}_p^*$, is replaced by the group of points on an elliptic curve defined over a finite field. Since the best algorithm known for solving the underlying com-

putationally hard mathematical problem, the *elliptic curve discrete logarithm problem* (ECDLP), takes fully exponential time, whereas the best algorithms known for solving the underlying computationally hard mathematical problems in RSA (integer factorization problem) and DSA (the discrete logarithm problem) take sub-exponential time, significantly smaller parameters can be used in elliptic curve cryptography (ECC) than in other systems such as RSA and DSA. For example, a 163-bit ECC key has a comparable level of security (against known attacks) as RSA and DSA with a modulus of 1024 bits [41]. This means by using ECC one can reach the same level of security with less expense of processing power, storage space, bandwidth and electrical power, which makes it especially interesting for applications on constrained devices such as smartcards, mobile phones and handhelds.

The performance of ECC depends mainly on the efficiency of finite field computations and fast algorithms for elliptic scalar multiplications. Selecting particular underlying fields and/or elliptic curves also speeds up the implementation. In Section 2.1, we already gave examples of such finite fields. Examples of families of curves that offer computational advantages are Koblitz curves over $\mathbb{F}_{2^m}$.

Let us now introduce the mathematical definition of the elliptic curves we will work with in the following. Suppose we have a finite field $\mathbb{F}_{2^m}$. Then, the polynomial equation

$$E : y^2 + xy = x^3 + ax^2 + b, \tag{2.1}$$

with coefficients $a, b \in \mathbb{F}_{2^m}$ together with the point at infinity $\mathcal{O}$ define an *elliptic curve* over $\mathbb{F}_{2^m}$. Let us ask for solutions (x, y) with $x, y \in \mathbb{F}_{2^m}$. Such a solution is called *point on the elliptic curve* E.

2.2.1 Arithmetic on General Elliptic Curves over $\mathbb{F}_{2^m}$

Elliptic Curve cryptography is based on multiplying a point P on an elliptic curve with a scalar k. This is nothing else than a k-fold addition of P. Hence, addition and scalar multiplication of points are important arithmetic operations, which for this reason will be described in the following.

Point Addition

There is an illustrative way to explain the law for adding two points on an elliptic curve. Let us assume that the coefficients of our elliptic curve E as well as x and y are rational numbers. Then, we can certainly draw the curve E which is the set of all solutions (x, y) of Equation (2.1).
Starting with two distinct rational points P and Q on E, we draw a line through P and Q and obtain a third point of intersection of the line with the curve. Reflecting this point in the x-axis, we obtain a point R, which we define to be the result of adding $P+Q$ (see Figure 2.1). Since the line, the curve, and the points of intersection are rational, R must also be rational.

Even if we only have one rational point P, we can still generally get another one by drawing the tangent line to the curve at P. This line will intersect with another point on the curve. If we then reflect this point in the x-axis we obtain the point $R = 2P$, which will also be rational (see Figure 2.2).

One can show that together with a zero element $\mathcal{O}$, these operations form a group. Let us agree on the convention that the points on our elliptic curve consist of the ordinary points in the ordinary affine x-y-plane together with a point $\mathcal{O}$ at infinity. We will call

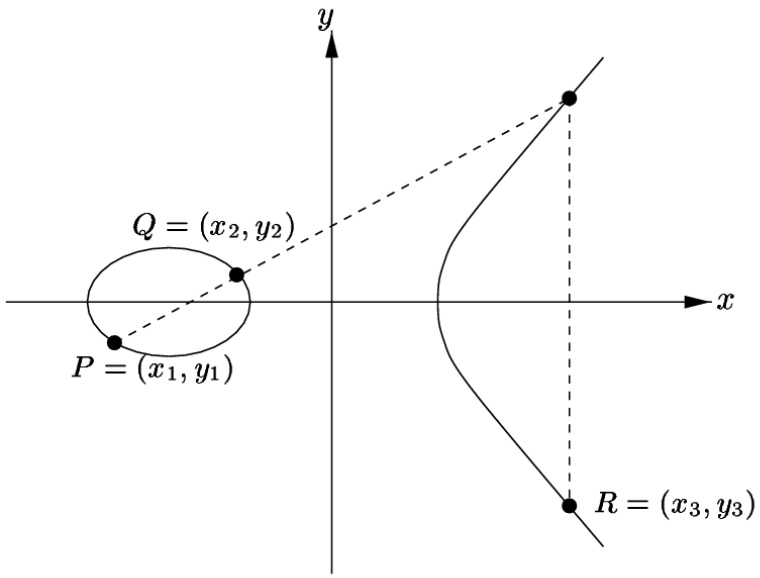


Figure 2.1: Adding two points on an elliptic curve [26].

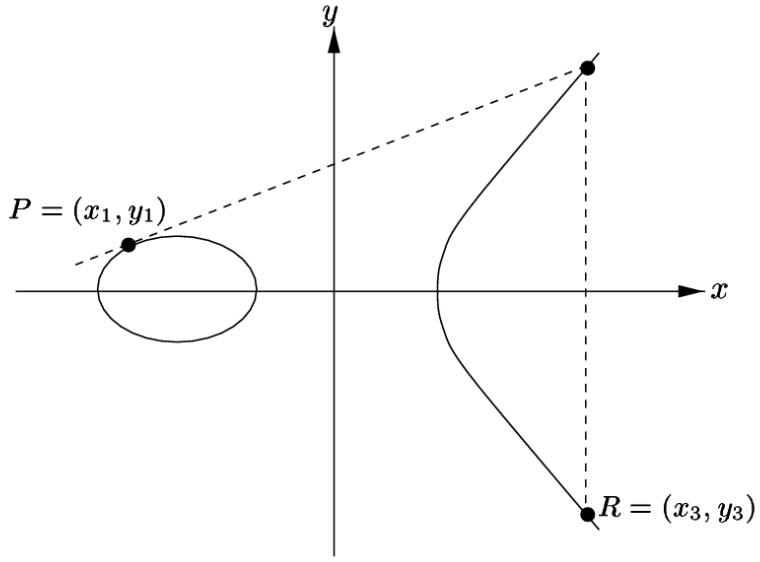


Figure 2.2: Doubling a point on an elliptic curve [26].

this point *point at infinity*.

Addition using affine coordinates Let us assume that $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ with $\mathcal{O} \neq P_1 \neq -P_2 \neq \mathcal{O}$ are two distinct points on the curve E given by Equation (2.1).

Based on the previously mentioned geometric motivation one can derive the following formulas [23] for computing the coordinates of the point $P_3 = P_1 + P_2 = (x_3, y_3)$. Note that we are in $\mathbb{F}_{2^m}$, i.e. $a, b, x_i, y_i \in \mathbb{F}_{2^m}$:

$$\begin{aligned} \lambda &= \begin{cases} \frac{y_1+y_2}{x_1+x_2} & \text{if } P_1 \neq P_2 \\ \frac{y_1}{x_1} + x_1 & \text{if } P_1 = P_2 \end{cases} \\ x_3 &= \lambda^2 + \lambda + x_1 + x_2 + a \\ y_3 &= (x_1 + x_3)\lambda + x_3 + y_1 \end{aligned} \tag{2.2}$$

The solutions for the two trivial cases $P_j = \mathcal{O}$ and $P_1 = -P_2$, which we have excluded above, are straightforward, namely $P_i + \mathcal{O} = P_i$ and $P_1 + (-P_2) = \mathcal{O}$, respectively.
In order to estimate and compare the computational cost of the following algorithms, which will use the above equations to add two points in affine coordinates, let us examine the equations with respect to the required number of field operations. Obviously, the formulas require 1 field division and 1 field multiplication. The computational cost of field additions and squarings can be neglected, since they can be done much faster than inversion or multiplication.

Addition using projective coordinates / mixed coordinates As inversion in $\mathbb{F}_{2^m}$ is computationally expensive relative to multiplication, it turns out that representing a point using projective coordinates might be advantageous. There exist several types of projective coordinates. In standard projective coordinates one finds that the projective point $(X, Y, Z), Z \neq 0$ corresponds to the affine point $(x, y) = (X/Z, Y/Z)$. Then, the projective equation of the elliptic curve is $Y^2Z + XYZ = X^3 + aX^2Z + bZ^3$.

Here are the computation steps for a projective point $P_1 = (X_1, Y_1, Z_1)$ and an affine

point $P_2 = (x_2, y_2)$ in the non-trivial case with $\mathcal{O} \neq P_1 \neq -P_2 \neq \mathcal{O}$. The result is the projective point $P_3 = P_1 + P_2 = (X_3, Y_3, Z_3)$.

$$\begin{array}{lllllllll} A & = & y_2 \cdot Z_1^2 + Y_1 & B & = & x_2 \cdot Z_1 + X_1 & C & = & Z_1 \cdot B \\ D & = & B^2 \cdot (C + aZ_1^2) & Z_3 & = & C^2 & E & = & A \cdot C \\ X_3 & = & A^2 + D + E & F & = & X_3 + x_2 \cdot Z_3 & G & = & X_3 + y_2 \cdot Z_3 \\ Y_3 & = & E \cdot F + Z_3 \cdot G & & & & & & \end{array} \tag{2.3}$$

This operation requires 10 field multiplications, which on most platforms is faster than addition in affine coordinates (1 field multiplication plus 1 field division).

Doubling a projective point, i.e. calculating $P_3 = P_1 + P_1$, can be done using the following formulas:

$$\begin{array}{lll} Z_3 & = & X_1^2 \cdot Z_1^2 \\ X_3 & = & X_1^4 + b \cdot Z_1^4 \\ Y_3 & = & bZ_1^4 \cdot Z_3 + X_3 \cdot (aZ_3 + Y_1^2 + bZ_1^4) \end{array} \tag{2.4}$$

This operation requires 5 field multiplications.

Scalar Point Multiplication

The computation of kP, where k is an integer and P is an elliptic curve point, is the basic operation of cryptographic schemes based on elliptic curves. It also dominates the execution time of those schemes. Thus, using efficient algorithms for point multiplication has a strong influence on the performance of elliptic curve cryptosystems. [23] presents a good overview over fast multiplication algorithms. In this thesis we only implement the binary method.

Binary method The simplest method for multiplying kP is based on the repeated-square-and-multiply method for exponentiation. Algorithm 1 shows how it works.

Algorithm 1 Left-to-right binary method for point scalar multiplication [23].
INPUT: $k = (k_{m-1}, \ldots, k_1, k_0)_2$, $P \in \mathbb{F}_{2^m}$.
OUTPUT: kP.
1: $Q \longleftarrow \mathcal{O}$.
2: **for** $i = m - 1$ downto 0 **do**
3: $\quad Q \longleftarrow 2Q$.
4: $\quad$ **if** $k_i = 1$ **then**
5: $\quad\quad Q \longleftarrow Q + P$.
6: $\quad$ **end if**
7: **end for**
8: **return** Q.

P is usually given in affine coordinates. In order to minimize the number of computationally expensive field inversions, we store Q in projective coordinates. The doubling in Step 3 is done as in Equations (2.4). For the addition in Step 5, we use Equations (2.3) to add P in affine coordinates to Q in projective coordinates.

Assuming that the average number of ones in the binary representation of k is $m/2$ and neglecting the fact that the very first point doubling is simply a doubling of $\mathcal{O}$, the algorithm requires approximately m point doublings and $m/2$ point additions. Using the computational complexity in terms of finite field operations derived for the point addition methods, we can also express the complexity of this multiplication method in terms of field multiplications and field divisions. The expected number of multiplications is $(5m + 10 \cdot (m/2)) = 10m$. Another multiplication and division is necessary to convert the result back to affine coordinates.

2.3 Introduction to Hyperelliptic Curves

In this section, we present an elementary introduction to the theory of hyperelliptic curves over finite fields of arbitrary characteristic, restricting attention to material that is relevant for this work. Material presented in this section was taken from [71]. For more details, the reader is referred to [30, 31, 42].

The idea that Jacobian groups of hyperelliptic curves (HEC) are suitable for cryptographical use was first introduced 1988 by Neal Koblitz [29]. Hyperelliptic curves are a special class of algebraic curves and can be viewed as generalization of elliptic curves. There are hyperelliptic curves of every genus $g \geq 1$. A hyperelliptic curve of genus $g = 1$ is an elliptic curve. Hence all the considerations regarding the DLP and the scalar multiplication for ECC lead with some modifications over to the HECC case. We will therefore only define hyper elliptic curves and introduce the group addition and group doubling used for our implementation.

Definition 2.3.1 *[42] Let $\mathbb{F}$ be a finite field, and let $\overline{\mathbb{F}}$ be the algebraic closure of $\mathbb{F}$. A hyperelliptic curve $\mathcal{C}$ of genus g over $\mathbb{F}$ ($g \geq 1$) is an equation of the form*

$$\mathcal{C} : v^2 + h(u)v = f(u) \qquad \text{in } \mathbb{F}[u, v], \tag{2.5}$$

where $h(u) \in \mathbb{F}[u]$ is a polynomial of degree at most g, $f(u) \in \mathbb{F}[u]$ is a monic polynomial of degree $2g + 1$, and there are no solutions $(u, v) \in \overline{\mathbb{F}} \times \overline{\mathbb{F}}$ which simultaneously satisfy the equation $v^2 + h(u)v = f(u)$ and the partial derivative equations $2v + h(u) = 0$ and $h'(u)v - f'(u) = 0$.

Figure 2.3 shows an example of a HEC $\mathcal{C} : v^2 = u^5 - 5u^3 + 4u + 3$.

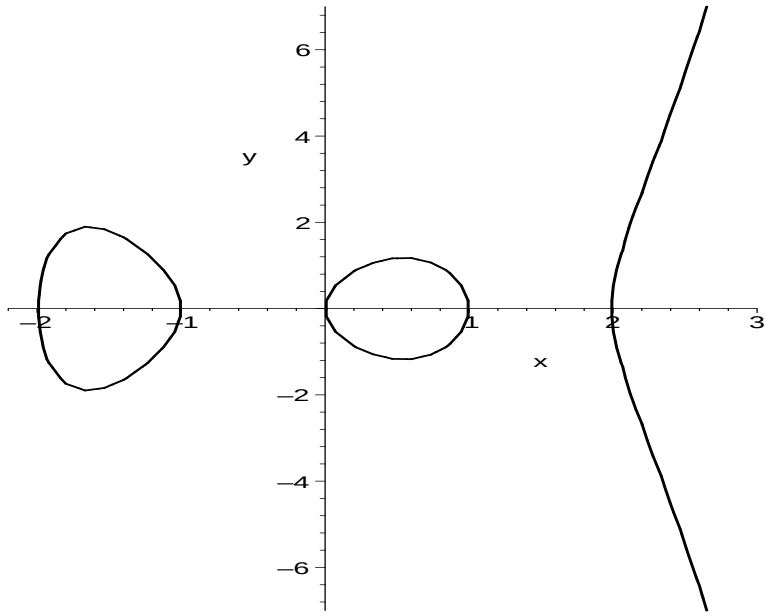


Figure 2.3: Hyperelliptic curve $\mathcal{C} : v^2 = u^5 - 5u^3 + 4u + 3$ over $\mathbb{R}$

2.3.1 Hyperelliptic Cryptosystems

The finite abelian groups obtained from the Jacobians of hyperelliptic curves can be used to construct public-key systems.

Discrete Log (DL) Problem: The DL Problem on $\mathbb{J}(\mathbb{K})$ is the problem, given two divisors $D_1, D_2 \in \mathbb{J}(\mathbb{K})$, of determining an integer k such that $D_2 = kD_1$, if such an k exists.

Computing Multiples of Divisors: A central ingredient in cryptosystems based on the DL problem in an abelian group is an efficient process for computing kD for $D \in \mathbb{J}(\mathbb{K})$ and for large integers k.

$$\underbrace{D + D + \cdots + D}_{\text{k times}} = kD$$

This operation is called *divisor multiplication* or *scalar multiplication*, and dominates the execution time of hyperelliptic cryptosystems. The simplest way to implement it is to use the binary method as presented in Algorithm 1 for ECC. The HECC version of this algorithm is slightly different from the ECC version. In the following we describe these differences.

For HECC the bit length of the scalar k is $2m$, where m is the extension of the underlying field $\mathbb{F}_{2^m}$. HECC works on divisors instead of points. The two main operations for HECC are the divisor addition (used in step 5 of Algorithm 1) and divisor doubling (used in step 3 of Algorithm 1).

The divisor P is usually given in affine coordinates. As we did in the case of ECC, we present the divisor Q in projective coordinates to minimize the number field inversions needed for a scalar multiplication. The following sections will deal with algorithms to perform the needed divisor operations efficiently.

About half of k's 2m bits are ones on average, hence the algorithm requires approximately 2m divisor doublings and m divisor additions for a scalar multiplication.

2.3.2 Cantor's Group Operations

Adding Divisors

The addition $D_1 + D_2$ of two divisors D_1 and D_2 will be calculated in two steps:

- **Composition Step:** First, find a semi-reduced divisor $D' = \mathrm{div}(a', b')$ such that D' is equivalent to $D_1 + D_2 = \mathrm{div}(a_1, b_1) + \mathrm{div}(a_2, b_2)$ in the group $\mathbb{J}$ (Algorithm 2).

- **Reduction Step:** Second, reduce the semi-reduced divisor

$D' = \mathrm{div}(a', b')$ to an equivalent reduced divisor $D = (a, b)$ (Algorithms 3 and 4).

Composition Step Algorithm 2 describes the Composition Step and was published by Cantor in 1987 [7].

Algorithm 2 Composition [7]

`Input:` Reduced divisors $D_1 = \mathrm{div}(a_1, b_1)$ and $D_2 = \mathrm{div}(a_2, b_2)$, both defined over $\mathbb{F}$
`Output:` A semi-reduced divisor $D' = \mathrm{div}(a', b')$ defined over $\mathbb{F}$ such that $D' \sim D_1 + D_2$

1. Use the Euclidean algorithm to find polynomials $d_1, e_1, e_1 \in \mathbb{F}[u]$ where $d_1 = \gcd(a_1, a_2)$ and $d_1 = e_1 a_1 + e_2 a_2$
2. Use the Euclidean algorithm to find polynomials $d, c_1, c_2 \in \mathbb{F}[u]$ where $d = \gcd(d_1, b_1 + b_2 + h)$ and $d = c_1 d_1 + c_2(b_1 + b_2 + h)$
3. Let $s_1 = c_1 e_1$, $s_2 = c_1 e_2$, and $s_3 = c_2$, such that $d = s_1 a_1 + s_2 a_2 + s_3(b_1 + b_2 + h)$
4. Set
   $a' = a_1 a_2 / d^2$
   and
   $b' = \frac{s_1 a_1 b_2 + s_2 a_2 b_1 + s_3(b_1 b_2 + f)}{d} \pmod{a'}$

Remark: The steps 1 to 3 of the algorithm can be written as a calculation of the greatest common divisor of three polynomials $d = \gcd(a_1, a_2, b_1 + b_2 + h) = s_1 a_1 + s_2 a_2 + s_3(b_1 + b_2 + h)$.

The proof that $D = \mathrm{div}(a, b)$ is a semi-reduced divisor and that $D \sim D_1 + D_2$ can be found in [18].

Reduction Step To complete the addition, a unique reduced divisor $D = \mathrm{div}(a, b)$ has to be found. There are two algorithms that are used for the reduction step: *Gauss reduction* and *Lagrange reduction* [16]. In the first algorithm the computation of the a_k (where a_k is the value of a in the iteration k of the algorithm) involves one multiplication

and one division of high degree polynomials. Each step is independent of the previous. However, as soon as a reduction step has been carried out, the formula for a_k can be rewritten using information from the previous step. Lagrange reduction takes advantage of this fact.

Algorithms 3 and 4 summarize Gauss and Lagrange reductions, respectively. Theorem 2.3.2 shows that the Gauss algorithm results in a reduced divisor.

Algorithm 3 Gauss reduction

`Input:` A semi-reduced divisor $D' = \mathrm{div}(a', b')$ defined over $\mathbb{F}$
`Output:` The (unique) reduced divisor $D = \mathrm{div}(a, b)$ such that $D' \sim D$

1. $a_0 = a'$, $b_0 = b'$
2. For $k = 1$ to t do (where t is minimal such that $\deg a_t \leq g$):
   2.1 $a_k = \frac{f - b_{k-1}h - b_{k-1}^2}{a_{k-1}}$
   2.2 $b_k = (-h - b_{k-1}) \bmod a$
3. Output $(a \leftarrow a_k, b \leftarrow b_k)$

Algorithm 4 Lagrange reduction

`Input:` A semi-reduced divisor $D = \mathrm{div}(a, b)$ defined over $\mathbb{F}$
`Output:` The (unique) reduced divisor $D' = \mathrm{div}(a', b')$ such that $D' \sim D$

1. $a_0 = a$, $b_0 = b$
2. $a_1 = \frac{f - b_0 h - b_0^2}{a_0}$
3. $-b_0 - h = q_1 a_1 + b_1$, with $\deg b_k \leq \deg a_k$
4. For $k = 2$ To t Do (where t is the minimal such that $\deg a_t \leq g$):
   4.1 $a_k = a_{k-2} + q_{k-1}(b_{k-1} - b_{k-2})$
   4.2 $-b_{k-1} - h = q_k a_k + b_k$, with $\deg b_k \leq \deg a_k$
5. Output $(a' \leftarrow a_k, b' \leftarrow b_k)$

Theorem 2.3.2 *[31] Let $D = \mathrm{div}(a, b)$ be a semi-reduced divisor. Then the divisor $D = \mathrm{div}(a, b)$ returned by Algorithm 3 is reduced, and $D' \sim D$.*

Doubling Divisors

The operation for doubling a divisor is easier since $a = a_1 = a_2$ and $b = b_1 = b_2$. Cantor's Algorithm for adding two equal divisors leads to algorithm 5.

Algorithm 5 Doubling

`Input:` Reduced divisors $D = \mathrm{div}(a, b)$ defined over $\mathbb{F}$
`Output:` A semi-reduced divisor $D' = \mathrm{div}(a', b')$ defined over $\mathbb{F}$ such that $D' \sim D + D$

1. Use the Euclidean algorithm to find polynomials $d, s_1, s_3 \in \mathbb{F}[u]$ where $d = \gcd(a, 2b + h) = s_1 a + s_3(2b + h)$
2. Set
   $a' = \frac{a^2}{d^2}$
   and
   $b' = \frac{s_1 ab + s_3(b^2 + f)}{d} \pmod{a}$

2.3.3 Harley's Group Operations

In [20], the authors noticed that one can reduce the number of operations by distinguishing between possible cases according to the properties of the input divisors. They described an efficient algorithm (using Karatsuba multiplication, CRT, and Newton Iteration) to reduce the overall complexity of the group operations.

To determine explicit formulae, it is essential to know the weight of the input divisor. The weight of a divisor is defined as the number of its points. For example, in the case of a hyperelliptic curve of genus 2, a divisor can have weight 0, 1, or 2. For each case,

implementations of different explicit formulae are required. In Figure 2.4, the main flow of the doubling algorithm of a genus-2 HEC is illustrated.

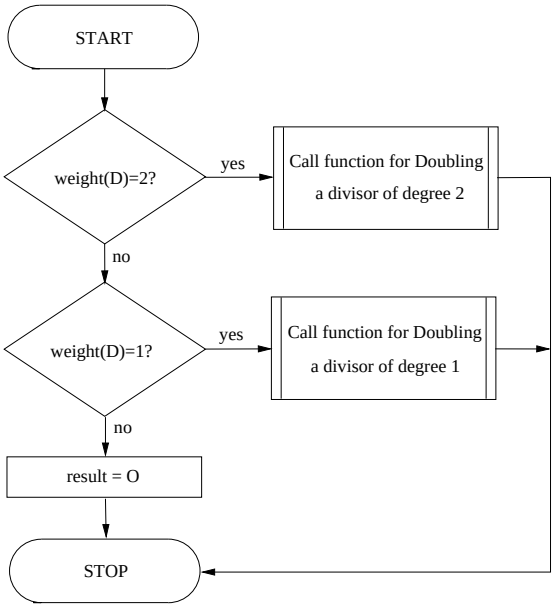


Figure 2.4: Main flow of Harley's doubling algorithm

Two polynomials have a linear factor in common with probability of $\approx 1/q$. Hence, in the case of a hyperelliptic curve of genus 2, we have no factor in common with the probability $P \approx 1 - 2^{-80}$. Therefore, in most cases the $gcd(u_1, u_2) = 1$. For the remainder of this thesis, we call this the *frequent case*. An implementation based on explicit formulae can be realized by avoiding the non frequent cases. This can be done on basis of a protocol which restarts if a non frequent case occurs. Without loss of generality (WLOG) we will consider only the cases for genus-2 curves in the following.

The Frequent Case of Doubling

The frequent case of doubling a divisor occurs if and only if $D = (u_1, v_1)$ *is of weight 2 and* $gcd(u_1, \tilde{h}) = 1$ *with* $\tilde{h} = h + 2v_1$*, i.e. both* u_1 *and* $\tilde{h}$ *do* not *have a factor in common.*

Algorithm 6 introduces the most frequent case for the group doubling considering arbitrary characteristic [35].

Algorithm 6 Frequent Case for Group Doubling (g=2)

Require: $D_1 = \mathrm{div}(u_1, v_1)$
Ensure: $D_2 = \mathrm{div}(u_2, v_2) = 2D_1$
1: $k = \frac{v_1^2 - v_1 h - f}{u_1}$ (exact division)
2: $s \equiv \frac{k}{h+2v_1} \bmod u_1$
3: $u' = s^2 + \frac{k - s(h+2v_1)}{u_1}$ (exact division)
4: $u_2 = u'$ made monic
5: $v_2 \equiv -(h + su_1 + v_1) \bmod u_2$

The Frequent Case of Adding

The frequent case of adding a divisor occurs if and only if $D_1 = (u_1, v_1)$ *and* $D_2 = (u_2, v_2)$ *are both of weight 2 and* $gcd(u_1, u_2) = 1$*, i.e. both* u_1 *and* u_2 *do* not *have a factor in common.*

Algorithm 7 introduces the group addition based on the ideas presented in [20] for arbitrary characteristic.

Algorithm 7 Frequent Case for Group Addition (g=2)

Require: $D_1 = \mathrm{div}(u_1, v_1)$, $D_2 = \mathrm{div}(u_2, v_2)$
Ensure: $D_3 = \mathrm{div}(u_3, v_3) = D_1 + D_2$
1: $k = \frac{f - v_1 h - v_1^2}{u_1}$ (exact division)
2: $s \equiv \frac{v_2 - v_1}{u_1} \bmod u_2$
3: $z = su_1$
4: $u' = \frac{k - s(z+h+2v_1)}{u_2}$ (exact division)
5: $u_3 = u'$ made monic
6: $v_3 \equiv -(h + z + v_1) \bmod u_3$

Inversion-Free Explicit Formulae

In [36, 37, 44, 71], the authors introduced a way to calculate the HEC group operations without using inversions. The authors in [44] were the first publishing this kind of approach. Their results were improved and generalized to even characteristic in [36, 37]. In [71], the up to date fastest formulae for inversion-free computation of HECC were introduced and this formulae were implemented in the work at hand. The group doubling and the mixed addition as introduced in [71] can be found in Table 2.1 and 2.2, respectively. For the remainder of this thesis we refer to *affine* coordinates, when talking about the usual representation. *Projective* denotes the inversion-free representation (in analogy to the elliptic curve cryptosystem).

In order to spare the inversion one has to introduce a coordinate Z. Hence, the two polynomial representing a divisor are given as following: $u(x) = x^2 + U_1/Zx + U_0/Z$ and $v(x) = V_1/Zx + V_0/Z$. We abbreviate a divisor in projective system as $[U_1, U_0, V_1, V_0, Z]$. In order to get a greater speed up for some scalar multiplication algorithm one can use mixed coordinates as inputs for the addition.

Table 2.1: Optimized inversionfree explicit formulae for adding a divisor on a HEC of genus two over $\mathbb{F}_{2^n}$ [71]

Input	$[U_{11}, U_{10}, V_{11}, V_{10}, Z_1]$; $[U_{21}, U_{20}, V_{21}, V_{20}, Z_2]$; $h = x$; and $f = x^5 + f_1 x + f_0$;	
Output	$[U_1', U_0', V_1', V_0', Z']$ $= [U_{11}, U_{10}, V_{11}, V_{10}, Z_1] + [U_{21}, U_{20}, V_{21}, V_{20}, Z_2]$	
Step	Procedure	Cost
1	precomputation: $Z = Z_1 Z_2; \tilde{U}_{21} = Z_1 U_{21}; \tilde{U}_{20} = Z_1 U_{20}; \tilde{V}_{21} = Z_1 V_{21}; \tilde{V}_{20} = Z_1 V_{20}$;	5M
2	compute resultant r of U_1, U_2: $z_1 = U_{11} Z_2 - \tilde{U}_{21}; z_2 = \tilde{U}_{20} - U_{10} Z_2; z_3 = U_{11} z_1 + z_2 Z_1$; $r = z_2 z_3 + z_1^2 U_{10}$;	6M + 1S
3	compute almost inverse of U_2 modulo U_1: $inv_1 = z_1; inv_0 = z_3$;	
4	compute s: $w_0 = V_{10} Z_2 - \tilde{V}_{20}; w_1 = V_{11} Z_2 - \tilde{V}_{21}; w_2 = inv_0 w_0$; $w_3 = inv_1 w_1; s_1 = (inv_0 + Z_1 inv_1)(w_0 + w_1) - w_2 - w_3(Z_1 + U_{11}); s_0 = w_2 - U_{10} w_3$;	8M
5	precomnputation: $R = Zr; s_0 = s_0 Z; s_3 = s_1 Z; \tilde{R} = R s_3; S_3 = s_3^2; S = s_0 s_1$; $\tilde{S} = s_3 s_1; \tilde{\tilde{S}} = s_0 s_3; \tilde{\tilde{R}} = \tilde{R}\tilde{S}$;	8M + 1S
6	compute l: $l_2 = \tilde{S}\tilde{U}_{21}; l_0 = S\tilde{U}_{20}; l_1 = (\tilde{S} + S)(\tilde{U}_{21} + \tilde{U}_{20}) - l_2 - l_0; l_2 = l_2 + \tilde{\tilde{S}}$;	3M
7	compute U': $U_0' = s_0^2 + s_1^2 z_1 (z_1 + \tilde{U}_{21}) + z_2 \tilde{S} + R(s_3 + r z_1); U_1' = \tilde{S} z_1 + R^2$;	6M + 2S
8	precomputations: $l_2 = l_2 - U_1'; w_0 = U_0' l_2 - S_3 l_0; w_1 = U_1' l_2 + S_3(U_0' - l_1)$;	4M
9	adjust $Z' = \tilde{R} S_3; U_1' = \tilde{R} U_1'; U_0' = \tilde{R} U_0'$;	3M
10	compute V' $V_0' = w_0 + \tilde{\tilde{R}} \tilde{V}_{20}; V_1' = w_1 + \tilde{\tilde{R}}(\tilde{V}_{21} + Z)$;	2M
Total		$45M + 4S$

Table 2.2: Optimized inversionfree explicit formulae for mixed adding a divisor on a HEC of genus two over $\mathbb{F}_{2^n}$ [71]

Input	$[U_{11}, U_{10}, V_{11}, V_{10}, 1]$; $[U_{21}, U_{20}, V_{21}, V_{20}, Z_{02}]$; $h = x$; and $f = x^5 + f_1x + f_0$;	
Output	$[U_1', U_0', V_1', V_0', Z'] =$ $[U_{11}, U_{10}, V_{11}, V_{10}, 1] + [U_{21}, U_{20}, V_{21}, V_{20}, Z_{02}]$	
Step	Procedure	Cost
1	precomputation: $\tilde{U}_{21} = U_{21}; \tilde{U}_{20} = U_{20}; \tilde{V}_{21} = V_{21}; \tilde{V}_{20} = V_{20};$	-
2	compute resultant r of U_1, U_2: $z_1 = U_{11}Z_2 - \tilde{U}_{21}; z_2 = \tilde{U}_{20} - U_{10}Z_2; z_3 = U_{11}z_1 + z_2; r = z_2z_3 + z_1^2U_{10};$	5M + 1S
3	compute almost inverse of U_2 modulo U_1: $inv_1 = z_1; inv_0 = z_3;$	
4	compute s: $w_0 = V_{10}Z_2 - \tilde{V}_{20}; w_1 = V_{11}Z_2 - \tilde{V}_{21}; w_2 = inv_0w_0; w_3 = inv_1w_1; s_1 = (inv_0 + inv_1)(w_0 + w_1) - w_2 - w_3(1 + U_{11}); s_0 = w_2 - U_{10}w_3;$	7M
5	precomnputation: $R = Z_2r; s_0 = s_0Z_2; s_3 = s_1Z_2; \tilde{R} = Rs_3; S_3 = s_3^2; S = s_0s_1; \tilde{S} = s_3s_1; \tilde{\tilde{S}} = s_0s_3; \tilde{\tilde{R}} = \tilde{R}\tilde{S};$	8M + 1S
6	compute l: $l_2 = \tilde{S}\tilde{U}_{21}; l_0 = S\tilde{U}_{20}; l_1 = (\tilde{S} + S)(\tilde{U}_{21} + \tilde{U}_{20}) - l_2 - l_0; l_2 = l_2 + \tilde{\tilde{S}};$	3M
7	compute U': $U_0' = s_0^2 + s_1^2z_1(z_1 + \tilde{U}_{21}) + z_2\tilde{S} + R(s_3 + rz_1); U_1' = \tilde{S}z_1 + R^2;$	6M + 2S
8	precomputations: $l_2 = l_2 - U_1'; w_0 = U_0'l_2 - S_3l_0; w_1 = U_1'l_2 + S_3(U_0' - l_1);$	4M
9	adjust $Z' = \tilde{R}S_3; U_1' = \tilde{R}U_1'; U_0' = \tilde{R}U_0';$	3M
10	compute V' $V_0' = w_0 + \tilde{\tilde{R}}\tilde{V}_{20}; V_1' = w_1 + \tilde{\tilde{R}}(\tilde{V}_{21} + Z_2);$	2M
Total		$38M + 4S$

3 Previous work

This chapter presents previous work that deals with the implementation of ECC and HECC. In this thesis we implemented ECC and HECC on an embedded platform using a software hardware codesign approach. Thus our emphasis lies on cryptosystems implemented in software and hardware on platforms with limited CPU power.

3.1 ECC over GF(2^m)

The performance of ECC over binary fields on FPGAs has been extensively studied in the last years. In Table 3.1 we compare a good number of ECC implementation papers using FPGAs and for comparison reasons we also list some performance numbers for ASICs, embedded and general processors as target platform, see Table 3.2. We included all relevant ECC implementations on FPGAs. We did not include for example [19], because this contribution only considers small fields and therefore is insecure for cryptographical use. In the following paragraphs we are going to describe in more detail the latest and fastest ECC implementations on FPGAs.

In [49], the authors introduced a processor architecture for elliptic curve cryptosystems over binary fields. The architecture is scalable in terms of area and speed and therefore can be optimized for different curves and fields. The processor consists of three main

components: the main controller, the arithmetic unit controller and the arithmetic unit. The Main controller interacts with the host system and controls the computation of kP. The arithmetic unit controller is responsible for the group operations, the coordinate conversions and controls the arithmetic unit (AU). The AU performs the field operations, like addition, squaring, multiplication and inversion. The most important field arithmetic components are the optimized bit-parallel squarer and a digit-serial multiplier, that can be reconfigured for different field orders and field polynomials. Using projective coordinates and the Montgomery scalar multiplication algorithm, they can calculate a scalar multiplication with arbitrary points in 0.21 ms considering the underlying field GF(2^{167}).

In [22], the authors present a programmable hardware accelerator to speed up point multiplication on elliptic curves over binary fields. The introduced architecture is scalable and handles curves with arbitrary fields up to $m = 255$. The authors also hardwired some of the commonly used curves to reach higher performance. They found out that most resources, namely about 73% of the LUTs and 46% of the flip-flops, are used by the multiplier, which clearly dominates the design in terms of hardware cost. In addition, the multiplication accounts for 62% of the execution time. Thus spending the biggest amount of the resources on it is justified. The performance could increased by using parallel execution and by separating control flow and data flow. Considering the hardwired curves over GF(2^{163}) they can perform a point multiplication 0.14 ms, this is to our knowledge the fastest result in the open literature.

platform		field & multiplication method	t_m [ms]
FPGA	Altera EPF10K250AGC599-2 @3MHz [47]	$GF(2^{163})$	
		Random: modified SSM multiplier	80.00
		Koblitz: modified SSM multiplier	45.00
	Xilinx XC4020XL@15MHz [68]	$GF(2^{155})$	18.40
	Xilinx XCV300-4@36MHz [38]	$GF(2^{155})$ binary-double-and-add	6.80
	Xilinx XC4062@16MHz [57]	$GF(((2^4)^2)^{21}, GF((2^8)^{21})$ Composite fields	4.50
	Xilinx XC4085XLA@37MHz [17]	$GF(2^{155})$ binary-double-and-add	1.30
	Xilinx XCV400E@76.7MHz [49]	$GF(2^{167})$ Montgomery (Digit Size D=16)	0.21
	Xilinx XCV2000E-FG680-7 @66.4MHz [22]	$GF(2^{163})$ Montgomery	0.14

Table 3.1: Timings for ECC scalar multiplications on different platforms

3.2 ECC over GF(p)

The emphasis in this thesis lies on implementing ECC over GF(2^m). Hence we just provide a short list of literature covering ECC implementations over GF(p) here [3, 6, 8, 12, 24, 25, 32, 50].

3.3 HECC Software Implementations on Embedded Platforms

Koblitz's idea to use HEC for cryptographic applications has been analyzed and implemented on several general purpose processors [15, 33, 51, 58–60, 65].

In [46] the author did the first implementation of HECC on an embedded processor namely using the FameXE smart card coprocessor. The author used genus-2 HEC and

platform		field & multiplication method	t_m [ms]
Embedded processor	Siemens SLE44C24S@5MHz [74]	$GF((2^8-17)^{17}) \approx GF(2^{134})$	
		binary-double-and-add	8370.00
		de Rooij w/18 precomputations	1830.00
	TI MSP430x33x@1MHz [21]	$GF(2^{128}-2^{97}-1) \approx GF(2^{128})$	
		binary-double-and-add (asm)	3400.00
	PDA Handspring Visor	$GF(2^{163})$	
	@16MHz [69]	Koblitz: $\tau - adic$	1670.00
	(Motorola Dragonball CPU)	Koblitz: $\tau - adic$ width-4	1510.00
		Koblitz: w/18 precomputations	870.00
general purpose processor	Sun UltraSPARC@300MHz [39]	$GF(2^{163})$ optimized Montgomery	
		w/o precomp.	10.50
	DEC Alpha 64-bit@175MHz [61]	$GF(2^{155})$	
		almost inv.	7.80
	Sun Fire 280R@900MHz [22]	$GF(2^{163})$	3.11
	PentiumII@400MHz [23]	$GF(32^{163})$	1.68
	PentiumII@200MHz [12]	$GF(32^{191})$	50
ASIC	Coprocessor VLSI@40MHz [1]	$GF(2^{155})$	3.90
	CE71 0,25μm 165k gates	$GF(2^{163})$	
	@66MHz [47]	Random: modified SSM multiplier	1.10
		Koblitz: modified SSM multiplier	0.65

Table 3.2: Timings for ECC scalar multiplications on different platforms

projective coordinates for the given implementation.

In [51] the author implemented for the first time HECC based on affine coordinates on embedded microprocessors, namely the ARM7TDMI and the PowerPC. Performance numbers are given for genus-2 and genus-3 HECC. The genus-3 group operation is based on the generalized and optimized explicit formulae presented in the same work.

The publication [53] introduces a low cost security implementation on the Pentium and the ARM processor. This contribution was a major step in accelerating genus-4 HECC by decreasing the complexity of the scalar multiplication by 72% compared to the operation by Cantor [7]. Performing arithmetic with 32-bit operands only, genus-4 HECC allow for a security comparable to the security of 128-bit ECC. Contrary to the general case, the implementation of genus-4 curves in groups of order $\approx 2^{128}$ outperforms implementations of genus-2 curves by a factor of about 1.5 . However, this last result does not hold anymore, because of recent improvements of genus-2 curves presented in [55].

In [52] the authors present improved and extended results of [51]. For certain genus-3 curves the presented group doubling formula saves more than 66% of the field multiplications compared to [34], resulting in a 52% improvement of the scalar multiplication. The genus-2 and genus-3 HECC implementations on an ARM7TDMI are compared to the ECC performance on the same platform. In this contribution the authors show that genus-2 curves are about a factor of 1.5 slower than ECC and that certain genus-3 curves perform the scalar multiplication at approximately the same time as ECC.

The results presented in [73] are the first thorough comparison of ECC and HECC on a wide range of important embedded platforms, namely ARM, ColdFire and PowerPC. The best timings for the scalar multiplication for HEC cryptosystems could be achieved on the PowerPC running at 50MHz, resulting in 117 and 84.9 milliseconds for genus 2 and 3, respectively. The scalar multiplication for ECC could be performed fastest on

the same platform in 106.3 ms. Table 3.3 reprints all the important implementation results. In [73] the authors conclude, similar to the the previously mentioned papers, that HECC allows for the same throughput as ECC and furthermore, in some cases HECC outperforms ECC. Additionally, the authors showed that the instruction cache on the PowerPC has a fundamental influence on the speed of scalar multiplications. The time needed to perform one scalar multiplication can be decreased by almost a factor of 8 when using instruction as well as data cache. A further speed up of 50% for the HEC scalar multiplication could be achieved by focussing on a fixed underlying field and curve.

A major improvement of the arithmetic on certain genus-2 hyperelliptic curves over

group order	platform	ECC	HECC		
			$g = 2$	$g = 3$	$g = 3, h(x) = 1$
$\approx 2^{160}$	ARM @ 50MHz	469.96	446.46	515.46	316.6
	ColdFire @ 90MHz	152.1	155.6	219.4	123.6
	PowerPC @ 50MHz	106.3	117	141.4	84.9
	Pentium @ 1.8GHz	2.6	3.61	4.15	2.58
$\approx 2^{170}$	ARM @ 50MHz	397.12	461.36	523.12	321.12
	ColdFire @ 90MHz	132.8	161.5	225.1	126.9
	PowerPC @ 50MHz	94.5	121.2	145.4	87
	Pentium 1.8 GHz	2.43	3.8	4.84	2.7
$\approx 2^{180}$	ARM @ 50MHz	515.95	516.5	577.5	356.99
	ColdFire @ 90MHz	171.7	183.4	246.7	146.2
	PowerPC @ 50MHz	121.8	138.1	160.1	96.8
	Pentium @ 1.8GHz	2.8	4.3	5.77	2.92
$\approx 2^{190}$	ARM @ 50MHz	436.01	542.68	581.24	360.24
	ColdFire @ 90MHz	157.8	187.6	258.5	147
	PowerPC @ 50MHz	112.4	141.7	167.8	101.8
	Pentium @ 1.8GHz	2.78	4.47	5.49	3.01

Table 3.3: Timings of the scalar multiplication of ECC and HECC on different platforms (in ms)

fields of characteristic two is presented in [55]. The authors were able to reduce the

number of multiplications for doubling a divisor class by approximately 47%, resulting in a performance gain of approximately 27% for HECC of genus 2 scalar multiplications. Hence the scalar multiplication for group orders between 2^{126} and 2^{190} on an ARM@80MHz takes 48.35 to 85.74ms. An updated version of the paper is presented in [54].

The latest contribution dealing with an implementation of HECC on embedded systems is presented in [71]. The author summarizes and updates in his thesis the results of previous papers. The performance numbers given are based on the newest and fastest explicit formulae mostly derived in the same publication. In [71], the author also analyzes software implementations on a general purpose processor, the parallelism of the group operation and the hardware implementation on FPGAs. Hence, he could put the implementation numbers on embedded processors in perspective with these other performance numbers. Therefore the author is able to show the practical relevance of HECC targeting a variety of implementations.

3.4 HECC Hardware Implementations

This section gives a short overview of the hardware implementations targeting HECC. The performance numbers are summarized in Table 3.4.

The first work discussing hardware architectures for the implementation of HECC appeared in [70,72]. The authors describe efficient architectures to implement the necessary field operations and polynomial arithmetic in hardware. All of the presented architectures are speed and area optimized. In [70], they also estimated that for a hypothetical clock frequency of 20 MHz, the scalar multiplication of HECC would take 21.4 ms using

the window NAF method.

In [5] the authors presented the first complete hardware implementation of a hyperelliptic curve coprocessor on a FPGA. This implementation targets a genus-2 HEC over $\mathbb{F}_{2^{113}}$. The target platform is a Xilinx II FPGA. Point addition and point doubling with a clock frequency of 45MHz take 105μs and 90μs, respectively. The scalar multiplication could be computed in 20.2 ms.

In [9, 10] the authors presented extended results of [5]. They implemented a HECC coprocessor using a variety of base fields, ranging from $\mathbb{F}_{2^{83}}$ to $\mathbb{F}_{2^{163}}$, as well as two different multipliers (digit size D=1 and D=4). The scalar multiplication takes between 9ms and 40ms and uses between 22,000 and 119,000 slices. The design options using D=4 multipliers presented in this work seem to have unreasonable hardware requirements.

Note that publications mentioned so far adopt the Cantor algorithm to compute group operations. Today exist more efficient algorithms to compute group addition and group doubling, as described in the previous sections.

The first approach to implement a hyperelliptic curve cryptosystem in hardware using explicit formulae is presented in [14]. They used the inversion-free group operations for HECC introduced in [36]. The results presented use a field GF(2^{113}), two different methods for the scalar multiplication (Binary Expansion Method and NAF) and two different digit multipliers ($D = 1$ and $D = 4$). They were able to reach a speed of 2.03ms for the best scalar multiplication. More details about the results presented in [14] are given in Table 3.4.

[27] presents for the first time a hardware implementation based on affine explicit HECC formulae. Keeping in mind that cryptographic implementations are always application specific, the authors provide three different designs. The designs of the HECC coprocessor ranging from high performance to low area. The high performance HECC coprocessor

is 74.2% faster than the best previous implementation and the low area implementation utilizes 82% less area than the smallest design published. Additionally, the authors analyze their design options by considering both the hardware requirements and the time constraints of the cryptographic application. Resulting in a better area-time product by a factor of 12 and 167 than previous publications. Through this improvement HECC is now approaching the performance range of ECC FPGA implementations.

		group order	Clock Cycles	Slices	Freq $[MHz]$	Time $[\mu s]$
[10]	affine coord.					
	binary (D=1)	2^{166}	-	22,000	-	10,000
	binary (D=4)	2^{166}	-	60,000	-	9,000
[14]	proj. coord.					
	binary (D=1)	2^{226}	339,057	22,183	45.6	7,530
	NAF (D=1)	2^{226}	332,913	21,550	45.6	7,390
	binary (D=4)	2^{226}	95,286	25,911	46.7	2,120
	NAF (D=1)	2^{226}	91,606	25,271	45.3	2,030
[27]	affine coord.					
Type 1	binary (D=27)	2^{162}	23,509	7,737	60.7	387
Type 2	binary (D=27)	2^{162}	34,012	5,674	51.4	662
Type 3	binary (D=27)	2^{162}	44,848	4,039	57.0	787

Table 3.4: Previous results of the HEC implementations on FPGA.

4 Tools and Hardware used

In this chapter we describe the development environment used. The target platform is the System-on-Chip® Lite (SoCLite) development system from NEC. Figure 4.1 shows a schematic with the two main functional blocks of the system.

The general purpose hardware block consists of the CPU and memory. The CPU is an ARM 7TDMI processor. In this thesis we refer to programs that run in this general purpose hardware block as software implementations.

The special hardware block in Figure 4.1 features a XILINX® VirtexE™ field programmable array (FPGA). This part of the board is the so called User Defined Logic (UDL). The UDL allows the user to implement custom logic by reconfiguring the FPGA. In this thesis we refer to any functions that are implemented in the special hardware block as hardware implementation.

With hardware software codesign we mean a mixture between a software and a hardware implementation. I.e. the main program runs on the ARM processor, while some routines of the program are executed by special hardware on the FPGA.

It is important to point out here that the FPGA does not have direct access to the main memory of the board (see Figure 4.1). Only the ARM processor can access the RAM directly.

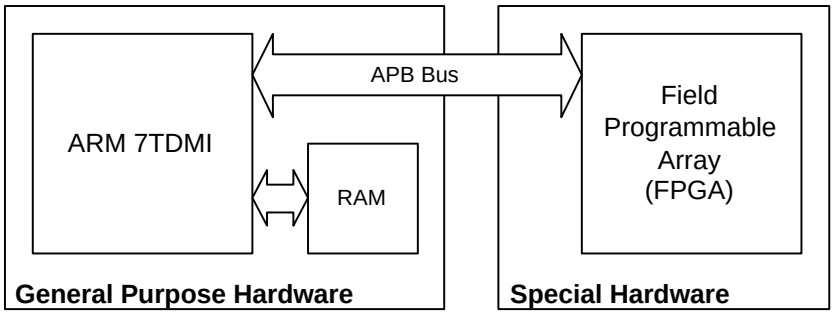


Figure 4.1: Schematic of SoCLite

4.1 Tools used

Here we describe briefly the tools used to program our testboard. The board is connected to a Standard IBM-compatible workstation as shown in Figure 4.2. All programming and compiling for the ARM processor is done on the workstation in standard C using the ARM Developer Suite V1.2. To download compiled programs to the ARM processor the ARM Multi-ICE (In Circuit Emulator) is used. The Multi-ICE connects the workstation's parallel port to the ARM's JTAG-Interface. Together with debugger software (e.g. ARM eXtended Debugger) the Multi-ICE allows the core of the processor to be started and stopped. This environment also allows the user to examine and modify registers and memory, and to set breakpoints and watchpoints within the software.

The FPGA is programmed in VHSIC Hardware Description Language (VHDL) using the Xilinx ISE 5.2i (Integrated Software Environment). As Hardware compiler we use XST, which is part of the ISE. To configure the FPGA the XILINX MultiLinx^{TM} Cable is used. The Cable is a peripheral hardware product used for downloading configuration and programming data to Xilinx FPGAs.

To receive results from the development board on a standard terminal program, the board is also connected to the workstation via a standard RS232 serial connection.

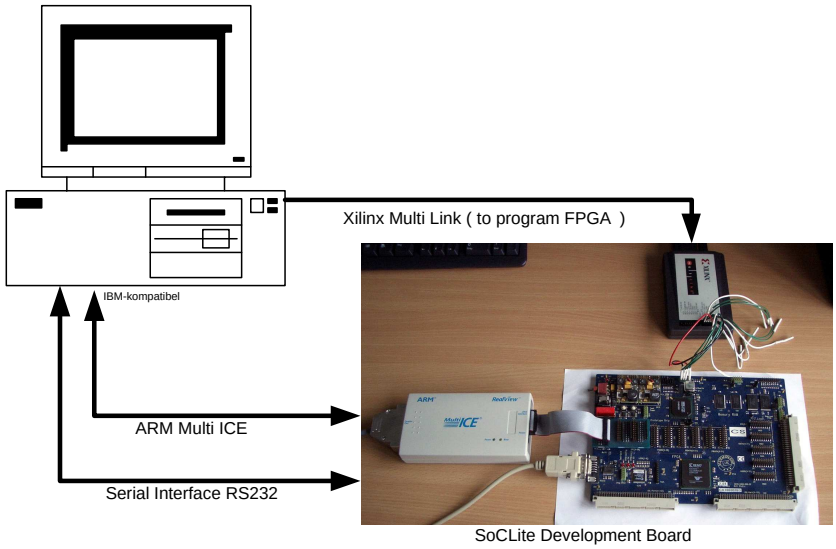


Figure 4.2: Setup SoCLite

4.2 SoCLite

In this section we describe the different functional blocks of our target platform in more detail. Figure 4.3 gives an overview of the different components of the SoCLite board we used. The figure also shows the bus structure that connects the different blocks on the board. For simplicity the figure only shows the blocks that are important for our thesis. For a more detailed overview of the SoCLite board see [45].

4.2.1 The ARM 7TDMI

The ARM 7TDMI is widely used processor core for embedded applications. It is based on ARM's version 4T 32/16-bit RISC (Reduced instruction set computer) architecture. The ARM 7TDMI on our board has the following features:

- *von Neumann architecture:* In this architecture data and program code share the

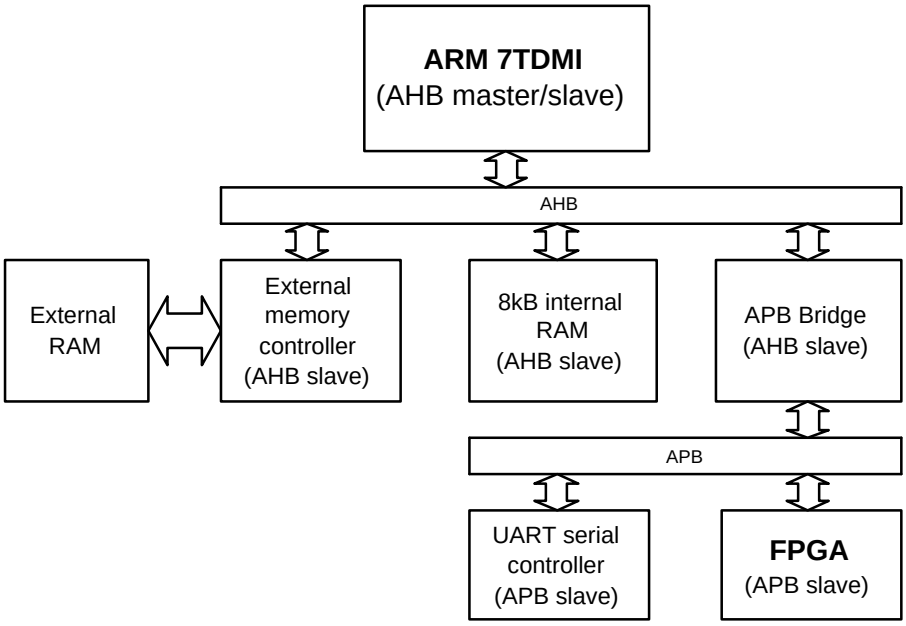


Figure 4.3: Block diagram of SoCLite

same memory.

- *load/store architecture:* This is a typical feature of RISC processors. All data processing is handled through registers. A direct manipulation of memory contents is not possible.

- Normal operation mode with a 32-bit ARM instruction set for maximum performance and flexibility.

- Thumb mode with a compressed 16-bit Thumb instruction set for increased code density.

- 32 Bit arithmetic logic unit (ALU)

- 32 Bit address space, which yields up to 4 GBytes of addressable memory space

- 8Kbyte of on-chip RAM

Besides the 8kB of on-chip RAM the SoCLite board also features 1Mbyte of SRAM which is available to the processor through an external memory interface (see Figure 4.3). The access to this external memory is slower than the on-chip memory.
During this thesis the ARM processor was always clocked at a frequency of 25MHZ.

4.2.2 The AMBA Bus System

The bus structure of the SoCLite is based on the Advanced Microcontroller Bus Architecture (AMBA). AMBA consists of two 32-Bit main buses as can be seen from Figure 4.3.
The Advanced High-performance Bus (AHB) is a high-speed multimaster bus to connect components like CPU and memory.
The Advanced Peripheral Bus (APB) is a simple lower speed bus to connect peripherals in an easy way. It is a single master bus with multiple slaves. As can be seen from 4.3 the APB is connected to the AHB through a gateway, the APB bridge, which is the master of the APB. The APB consists of a 32 Bit data bus, an address bus and three control signals (select, write and enable) to transfer data to/from devices. The select signal tells the device that it has been selected for the next data transfer. If the write signal is set, the device knows that data is written to it, otherwise, if write is not set, data is read from the device. In case of a read access the enable signal tells the device when it actually has to strobe data on the read data bus. In case of a write access the enable signal tells the device when to sample data from the write data bus. A read or write access on the APB bus takes 4 clock cycles, when there are no wait states inserted, which is the case in our work. For more detailed information on AMBA and APB refer to [2] [45].

4.2.3 The Field Programmable Array (FPGA)

The FPGA on our testboard is a Xilinx® VirtexE XCV2000E™. It features 19200 slices, with two Flip Flops and two 4 input Look-Up-Tables (LUTs) on each slice. It has 160 Blocks of select RAM, which can hold up to 655,360 Bits of Data together. The maximum frequency at which the FPGA can be clocked on our board is 115MHz, but we clocked the FPGA always at 25MHz, the same frequency as the ARM processor.
The FPGA is connected to the ARM processor as an APB Slave. The Address bus connection is 26 Bit wide, which equals 512MBytes of available address space in the memory map of the ARM processor. The read and write Data busses are 32 Bits wide each.

5 Implementation of Finite Field Arithmetic over $\mathbb{F}_{2^m}$

Figure 5.1 shows the hierarchy for a scalar multiplication of ECC/HECC. At the top most level of the pyramid is the scalar multiplication $k * P$. The scalar multiplication depends on the group operations for doubling and addition, which reside one level lower. The group operations in turn depend on the field arithmetic. Hence the field arithmetic functions build the foundation of ECC and HECC. In this chapter we describe how we implemented these $\mathbb{F}_{2^m}$ field arithmetic functions in software and in hardware.

In all of our ECC implementations m will be $m = 167$, where m is the extension of underlying field $\mathbb{F}_{2^m}$. An HECC of genus 2 with $m = 81$ provides a comparable level of security as does a ECC with $m = 167$. Hence we choose m to be $m = 81$ in all of our HECC implementations.

Before we implemented the field arithmetic on our embedded platform, we created a reference implementation of the scalar multiplication for ECC and HECC. These reference implementations were programmed using NTL5.3 [62] and Microsoft Visual C++ 6.0 on a PC. We used them to generate test vectors to check the results of our implementations on the target platform.

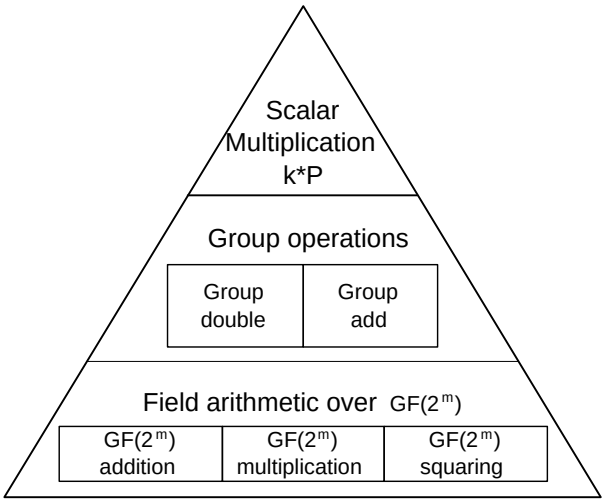


Figure 5.1: Architecture of elliptic/hyperelliptic cryptosystems

5.1 Implementation in Software

By software implementation we mean code that runs entirely on the ARM 7TDMI processor and that does not use FPGA at all. Our software implementations are programmed using standard C.

The target CPU used in [56] is based on the ARM V4 architecture, which is similar to our target CPU. Thus we use the C code from that thesis as a basis for our implementation of the field arithmetic. We adapt the code for our needs, e.g. we use $m = 81$ and $m = 167$ instead of $m = 163$.

The following explanations and implementation of finite field arithmetic are mostly based on [56], [23] and [69].

5.1.1 Field representation

As mentioned in [23] and in Section 2.1.2 of this document, the polynomial basis representation of $\mathbb{F}_{2^m}$ with a trinomial or pentanomial as a reduction polynomial $f(x)$ appears

to yield the simplest and fastest implementation in software. Therefore, we use this representation throughout our implementation of finite field arithmetic.

Consequently, we write an element $A \in \mathbb{F}_{2^m}$ as polynomial $A(x) = \sum_{i=0}^{m-1} a_i x^i$ and store it as binary vector $A = (a_{m-1}, \ldots, a_0)$. How to store these coefficients in computers is straightforward: since they are binary values, one can store each coefficient in one bit of memory. Due to the 32-bit register architecture of the ARM CPU, we use an array of $s = \lceil m/32 \rceil$ 32-bit words $(A[s-1], \ldots, A[0])$. The rightmost bit of $A[0]$ corresponds to a_0 and a_{m-1} is part of $A[s-1]$. The bits left of a_{m-1} are set to zero. I.e. for our ECC implementation with $m = 167$ an array of $s = 6$ 32-Bit words is sufficient. For our HECC implementation with $m = 81$ an array of $s = 3$ words is sufficient.

We basically defined two types, namely `gf2mShortElement` and `gf2mLongElement`, that are arrays of s and $2s$ words. The later type holds intermediate results of multiplications or squarings, while the first one is used for all other field elements.

5.1.2 Addition

As stated in Section 2.1.2, addition of field elements is performed by bitwise XOR operations. Thus the addition of two field elements requires 6 XOR concatenations of 32-Bit words for ECC and 3 concatenations for HECC.

5.1.3 Multiplication

The product $c = a \cdot b$ is computed by first evaluating $c'(x) = a(x) \cdot b(x)$ followed by modular reduction $c(x) \equiv c'(x) \mod f(x)$.

Polynomial Multiplication

To our knowledge, Algorithm 8 is the fastest method to compute $c = a \cdot b$. It does this by

using a window method [40]. In Step 1, polynomials $B_u = u(x) \cdot b(x)$ are precomputed for $0 \leq u \leq 2^w$ where w is the window size. This is done using a couple of left shifts and cumulative field element additions. In Steps 6 and 10 the m-bit vector B_u is added to C where the rightmost bit of B_u is added to the rightmost bit of $C\{j\}$. Here, $C\{j\}$ denotes the bit vector $(C[2s-2], \ldots, C[j])$ and the symbol $\ll$ denotes a bitwise left shift. The result of the algorithm is not reduced, so $c(x)$ is of degree at most $2m-2$.

Algorithm 8 Left-to-right comb method with windows of width $w = 4$ [23]

INPUT: Binary polynomials $a(x)$ and $b(x)$ of degree at most $m-1$
OUTPUT: A binary polynomial $c(x) = a(x) \cdot b(x)$ of degree at most $2m-2$

1: Compute $B_u = u(x) \cdot b(x)$ for all polynomials $u(x)$ of degree at most 3.
2: $C \longleftarrow 0$.
3: **for** $k = 7$ downto 1 **do**
4: **for** $j = 0$ to $s-1$ **do**
5: Let $u = (u_3, u_2, u_1, u_0)$, where u_i is bit $(4k+i)$ of $A[j]$.
6: $C\{j\} \longleftarrow C\{j\} \oplus B_u$.
7: **end for**
8: $C\{j\} \longleftarrow C\{j\} \ll 4$.
9: **end for**
10: **for** $j = 0$ to $s-1$ **do** {The case $k = 0$. This saves one comparison in the loop above.}
11: Let $u = (u_3, u_2, u_1, u_0)$, where u_i is bit i of $A[j]$.
12: $C\{j\} \longleftarrow C\{j\} \oplus B_u$.
13: **end for**
14: **return** $c(x)$.

Modular Reduction

By choosing the reduction polynomial $f(x)$ to be a low weight polynomial, i.e. one with the least possible number of non-zero coefficients, reduction modulo $f(x)$ becomes a computationally cheap operation [4]. For cases of practical interest, $f(x)$ is a trinomial or pentanomial, i.e. the number of non-zero coefficients is 3 or 5. Reduction of $c(x)$ modulo $f(x)$ can be efficiently performed one word at a time. Suppose $f(x) = x^{167} + x^6 + 1$.

Then

$$\begin{aligned} x^{167} &\equiv x^6 + 1 \mod f(x) \\ &\vdots \\ x^{288} &\equiv x^{127} + x^{121} \mod f(x) \\ &\vdots \\ x^{332} &\equiv x^{171} + x^{165} \mod f(x) \end{aligned}$$

Hence, reduction can be performed by adding to C properly aligned all those $C[j]$ that contain the coefficients $c_k, 167 \leq k \leq 332$.

Let us clarify this with an example. Suppose we have already reduced C[10] and now want to reduce C[9], which contains bits 288 through 319. We can write the corresponding polynomial as

$$\begin{aligned} c^{(9)}(x) &= \sum_{k=0}^{31} c_{k+288} \cdot x^{k+288} = x^{288} \sum_{k=0}^{31} c_{k+288} \cdot x^k \\ &\equiv (x^{127} + x^{121}) \cdot \underbrace{\sum_{k=0}^{31} c_{k+288} \cdot x^k}_{C[9]} \mod f(x) \end{aligned}$$

Now, we add the coefficients of x^i in $c^{(9)}(x)$ to the corresponding coefficient of x^i in C. Figure 5.2 shows how $C[9]$ is aligned and how the words $C[5]$ through $C[3]$ of C are affected. The result of these thoughts is Algorithm 9. The left shift of bits is denoted by $\ll$ and the right shift by $\gg$.

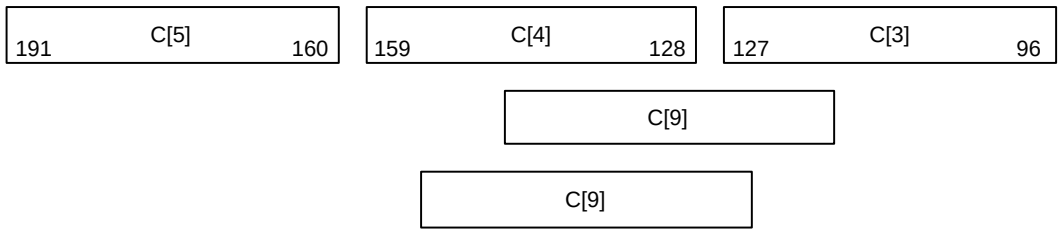


Figure 5.2: This figure shows how the two shifted versions of $C[9]$ are aligned before being added to C.

Algorithm 9 Modular reduction in $\mathbb{F}_{2^m}$ by $f(x) = x^{167} + x^6 + 1$ [23]

INPUT: A binary polynomial $c(x)$ of degree at most $2m - 2 = 332$
OUTPUT: A binary polynomial $c(x)$ mod $f(x)$ of degree at most $m - 1$

1: **for** $i = 10$ downto 6 **do** {Reduce $C[i]$ modulo $f(x)$}
2: $T \longleftarrow C[i]$.
3: $C[i-6] \longleftarrow C[i-6] \oplus (T \ll 31) \oplus (T \ll 25)$.
4: $C[i-5] \longleftarrow C[i-5] \oplus (T \gg 1) \oplus (T \gg 7)$.
5: **end for**
6: $T \longleftarrow C[5]$ **and** 0xFFFFFF80.
7: $C[0] \longleftarrow C[0] \oplus (T \gg 1) \oplus (T \gg 7)$.
8: $C[5] \longleftarrow C[5]$ **and** 0x0000007f. {Clear the unused bits of $C[5]$.}
9: **return** $(C[5], C[4], C[3], C[2], C[1], C[0])$.

5.1.4 Squaring

Squaring in $\mathbb{F}_{2^m}$ can be done much faster than multiplying two arbitrary elements. It turns out to be a linear operation, because for $a(x) = \sum_{i=0}^{m-1} a_i x^i$ one finds

$$a^2(x) = \sum_{i=0}^{m-1} \sum_{j=0}^{m-1} a_i a_j x^{i+j} = \sum_{i=0}^{m-1} a_i x^{2i}$$

Hence, squaring can be done by simply inserting a 0-bit between consecutive bits of the binary representation of a. A fast way to do that is to use a table of precomputed values as shown in Algorithm 10 [61], which is a static table in our implementation. Hence, the precomputation step needs not to be done during run-time. The reduction step is done

using Algorithm 9.

Algorithm 10 Squaring in $\mathbb{F}_{2^m}$ [61]

INPUT: A binary polynomial $a(x)$ of degree at most $m-1$
OUTPUT: A binary polynomial $b(x) = a^2(x) \mod f(x)$ of degree at most $m-1$

1: Precompute for each byte $v = (v_7, \ldots, v_1, v_0)$ the 16-bit vector $T(v) = (0, v_7, \ldots, 0, v_1, 0, v_0)$.
2: **for** $i = 0$ to $s-1$ **do**
3: Let $A[i] = (u_3, u_2, u_1, u_0)$ where each u_j is a byte.
4: $C[2i] \longleftarrow (T(u_1), T(u_0))$, $C[2i+1] \longleftarrow (T(u_3), T(u_2))$.
5: **end for**
6: Compute $b(x) = c(x) \mod f(x)$. {using Algorithm 9.}
7: **return** b(x).

5.1.5 Optimizations

We use a different compiler than [56] and our target platform is, in terms of memory and CPU performance, more constrained than the PDA used in [56]. Hence we use three different approaches to increase the performance of the field arithmetic functions on our target platform.

Our target platform has 8 kByte on-chip memory, which can be accessed faster than the 1MByte external RAM. Thus we have to make sure that the compiler locates all variables and the stack within the internal RAM. When doing this we have to ensure that there won't be any stack collisions, during program execution though.

The field arithmetic function gf2Mult is much slower than gf2mSquare and gf2mAdd. It therefore accounts for the biggest time share when executing a scalar multiplication. Hence we choose to put some optimization effort into gf2mMult. The first step to get a lower execution time for the multiplication is to unroll all loops in the function.

Replacing all functions that are marked as inline functions manually yields a further

increase in performance. Usually this manual replacement is not necessary, since the compiler should replace these functions automatically. But the compiler we use does not seem to be able to replace these functions correctly on its own.

By applying the two optimizations we could cut down the execution time of the field multiplication to 68% of the time without optimizations. Table 5.1 gives an overview of the timings for the field arithmetic functions with $m = 167$ and $m = 81$ after the optimizations.

Function name	$m = 167$ [μs]	$m = 81$ [μs]
gf2mAdd	8	3
gf2mSquare	34	20
gf2mMult	450	194

Table 5.1: Timings for $\mathbb{F}_{2^m}$ functions in software @25MHz

5.2 Implementation in Hardware

In this section we show how we implemented the three field arithmetic functions on the FPGA. Table 5.4 shows the timings of our $\mathbb{F}_{2^m}$ field arithmetic implementations in hardware.

The advantage when creating specialized hardware is that we do not have to stick to the limitations of general purpose hardware like the ARM 7TDMI. For example we do not have the limitation of 32-Bit wide registers. Hence we choose the width of the registers in our hardware implementations to be equal to the length of a field element, i.e. for ECC 167 Bits and for HECC 81 Bits.

5.2.1 The Adder

The addition of two $\mathbb{F}_{2^m}$ field elements is a simple bitwise XOR between the two input field elements. The two input elements are stored in their own registers. The Adder XORs all the bits of the input registers at once and stores the result in an output register. Hence an $\mathbb{F}_{2^m}$ addition in hardware takes one clock cycle for ECC and HECC.

5.2.2 The Multiplier

In this thesis we use the implementation of a Least Significant Digit (LSD) first multiplier presented in [48]. The concept of this LSD multiplier was introduced by [66]. It computes the multiplication of two $\mathbb{F}_{2^m}$ elements A and B in m/D clock cycles, where m is the extension of the underlying field and D the digitsize of the multiplier. This means that for a higher m the number of clock cycles, a multiplication takes, goes up and for a higher D the number goes down. We have to keep in mind though that the hardware complexity of the LSD implementation increases for bigger m's and bigger D's. There are two extreme cases of the LSD multiplier. For $D = 1$ the multiplier would be a fully serial multiplier, which would need m clock cycles for a multiplication and would have a very low hardware complexity. For $D = m$ the multiplier would be a fully parallel multiplier, which computes a multiplication in one cycle. But the hardware complexity required for such a D would be very high. Hence the choice of the digitsize is a tradeoff between performance and hardware complexity. We implemented the following digitsizes $D = 4,\ 8,\ 16\ and\ 32$ in this thesis.

Figure 5.3 shows a block diagram of the LSD. It consists of 4 functional blocks: the "Ax^{Di} mod $F(x)$" block, the digit multiplication core, the accumulator, and the "$mod\ F(x)$" block.

For each clock cycle i the "$Ax^{Di} \mod F(x)$" block generates a term $A^{(i)} = A^{(i-1)}x^D \mod F(x)$

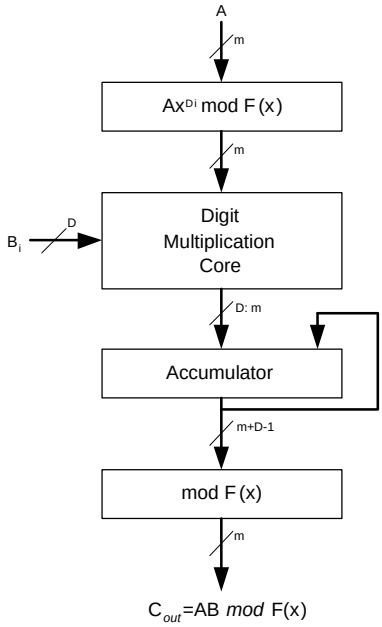


Figure 5.3: Least Significant Digit first $\mathbb{F}_{2^m}$ Multiplier [48]

with $i = 1 \ldots [\frac{m}{D} - 1]$ and $A^{(0)} = A$. Equation 5.1 defines $A^{(i)}$ for optimum primitive polynomials [67]. These are polynomials of the form $F(x) = x^m + \sum_{j=0}^{t} f_i x^i$, where $m - t \geq D$.

$$A^{(i)} = x^D \sum_{j=0}^{m-D-1} a_j^{(i-1)} + (\sum_{m-D+j}^{(i-1)} x^j)(\sum_{j=0}^{t} f_j x^j) \tag{5.1}$$

The digit multiplication core computes a set of D scalar products at each clock cycle, which is then passed on to the accumulator. The sum of these products $b_{Di}A^{(i)}$, $b_{Di+1}A^{(i)}x$, ..., $b_{Di+D-1}A^{(i)}x^{D-1}$ represents the product $A^{(i)}B_i$ as shown in equation 5.2.

$$A^{(i)}B_i = \sum_{j=0}^{D-1} A^{(i)} b_{Di+j} x^j \tag{5.2}$$

The accumulator adds up the D scalar values and the value, stored in it from the previous clock cycle.

The "*mod* $F(x)$" functional block reduces the accumulated value of maximum degree $m + D - 1$ as shown in Equation 5.3. This equation is valid for optimum primitive polynomials.

$$C_{out} = \sum_{j=0}^{m-1} c_j x^j + (\sum_{j=0}^{D-2} c_{m+j} x^j)(\sum_{k=0}^{t} f_k x^k) \tag{5.3}$$

The implementation of the LSD only accepts trinomials as reduction polynomials. The reduction polynomials for $m = 81$ and $m = 167$ are trinomials, thus the limitation to trinomials is no problem for our implementations.

5.2.3 The Squarer

We implement two different squarers in this thesis, one for the ECC implementation with $m = 167$ and one for HECC with $m = 81$. According to [70] squaring in $\mathbb{F}_{2^m}$ inclusive reduction of the result can be done in one clock cycle by connecting the appropriate input bits, via XOR, to make up the output bits. Tables 5.2 and 5.3 show how the input bits I_n are mapped to the output bits O_n for $m = 167$ and $m = 81$ respectively. The XOR operation will be denoted by $\oplus$.

$$
\begin{aligned}
O_0 &\longleftarrow I_0 \oplus I_{164} \\
O_1 &\longleftarrow I_{84} \\
O_2 &\longleftarrow I_1 \oplus I_{165} \\
O_3 &\longleftarrow I_{85} \\
O_4 &\longleftarrow I_2 \oplus I_{166} \\
O_5 &\longleftarrow I_{86} \\
O_6 &\longleftarrow I_3 \oplus I_{164} \\
O_8 &\longleftarrow I_4 \oplus I_{165} \\
O_{10} &\longleftarrow I_5 \oplus I_{166} \\
O_{2n+1} &\longleftarrow I_{n+84} \oplus I_{n+85} \ \forall\ 3 \leq n \leq 82 \\
O_{2n} &\longleftarrow I_n \ \forall\ 6 \leq n \leq 83
\end{aligned}
$$

Table 5.2: Squarer outputs for $m = 167$

$$
\begin{aligned}
O_0 &\longleftarrow I_0 \oplus I_{79} \\
O_1 &\longleftarrow I_{41} \\
O_2 &\longleftarrow I_1 \oplus I_{80} \\
O_3 &\longleftarrow I_{42} \\
O_4 &\longleftarrow I_2 \oplus I_{79} \\
O_6 &\longleftarrow I_3 \oplus I_{80} \\
O_{2n+1} &\longleftarrow I_{n+39} \oplus I_{n+41} \ \forall\ 2 \leq n \leq 39 \\
O_{2n} &\longleftarrow I_n \ \forall\ 4 \leq n \leq 40
\end{aligned}
$$

Table 5.3: Squarer outputs for $m = 81$

Function in hardware	$m = 167$ [clock cycles]	$m = 81$ [clock cycles]
$\mathbb{F}_{2^m}$ addition	1	1
$\mathbb{F}_{2^m}$ square	1	1
$\mathbb{F}_{2^m}$ multiplication with $D = 4$	42	21
$\mathbb{F}_{2^m}$ multiplication with $D = 8$	21	11
$\mathbb{F}_{2^m}$ multiplication with $D = 16$	11	6
$\mathbb{F}_{2^m}$ multiplication with $D = 32$	6	3

Table 5.4: Timings for $\mathbb{F}_{2^m}$ functions in hardware

6 Design options

In this chapter we present the 4 different design options that we implemented in this thesis. Each design option was implemented for ECC and HECC. In the following we also refer to the different design options as Type I, II, III, and IV.

The main aim in embedded systems is to get as much performance out of limited hardware as possible in order to keep the cost as low as possible. With the different design options we show how much performance gain can be achieved by adding a certain amount of extra hardware to an architecture.

The Type I architecture is a software only implementation, i.e. it runs entirely on the ARM 7TDMI and does not use any special hardware.

The Type II, III and IV options reflect our approach to hardware software codesign. In these design options we implement functions in hardware that in Type I are implemented in software. In every design option we implement more software functionality in hardware, e.g. Type III implements more functionality in hardware than Type II and so on.

6.1 Type I: Software only

The Type I design option is running on the ARM 7TDMI only and does not use the FPGA at all. In this section we show how we implemented the group operations and the scalar multiplication for ECC and HECC on the target platform. The parts describing the implementation of the ECC group operations are taken in part from [56]. The implementation of the field arithmetic in software was already described in chapter 5.1.

6.1.1 Implementation of Elliptic Curve Group Operations

In this section, we describe the implementation of the elliptic curve group operations in more detail.

Definitions

For our implementation, we define the points with the affine coordinates $(0, 0)$ and with the projective coordinates $(\cdot, \cdot, 0)$, i.e. the projective points with $z = 0$, to be the point at infinity $\mathcal{O}$.

Point Addition using mixed coordinates

Adding a point given in affine coordinates to a point given in projective coordinates can be realized by simply implementing Equations (2.3). Figure 6.1 gives an overview of the different special cases we distinguish to speed up execution. These cases are

1. $P_{\text{proj}} = -P_{\text{aff}}$ iff $X_1 = x_2 \cdot Z_1$ and $Y_1 = (x_2 + y_2) \cdot Z_1^2$,
2. $P_{\text{proj}} = P_{\text{aff}}$ iff $X_1 = x_2 \cdot Z_1$ and $Y_1 = y_2 \cdot Z_1^2$.

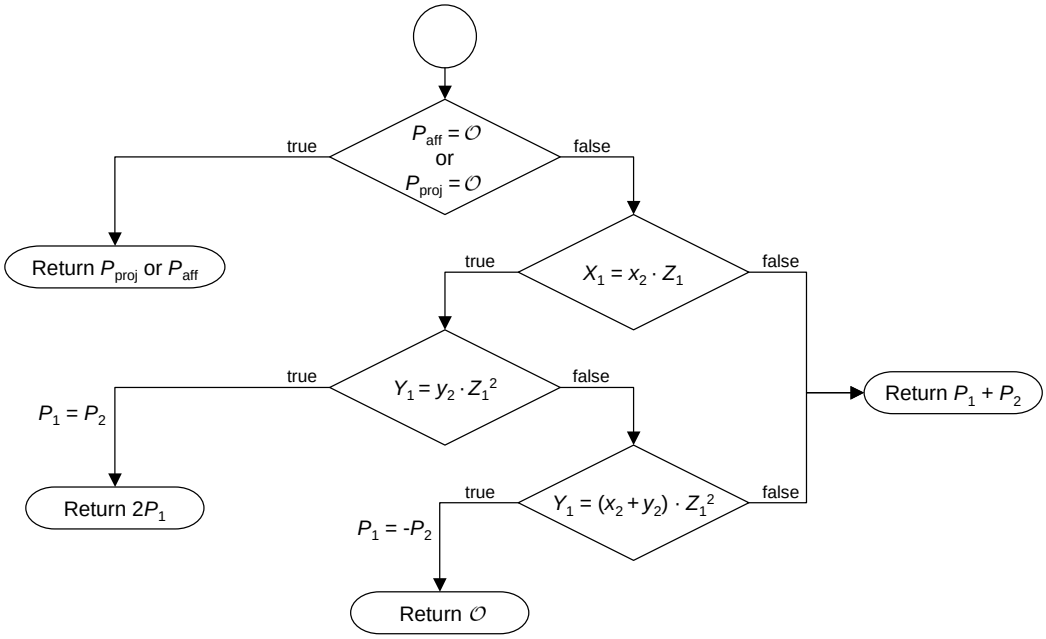


Figure 6.1: Flow diagram of the decision process to distinguish the special cases for point addition in mixed coordinates [56].

Point doubling using projective coordinates

Point Doubling is a special case of Point addition. The realization in software is a straight forward implementation of Equations 2.4.

6.1.2 Implementation of Hyperelliptic Curve Group Operations

This section covers the implementation of the hyperelliptic curve arithmetic in more detail. The underlying finite field arithmetic is the same for elliptic and hyperelliptic curves, with the difference that for our elliptic curves we chose $m = 167$ and for our hyperelliptic curves we chose $m = 81$. The smaller size of m increases the performance of the field arithmetic functions considerably. But this performance gain in field arithmetic is used up due to a more computational expensive group arithmetic for hyperelliptic

curves.

Elliptic curve group arithmetic operates on points consisting of two m Bit in affine and three m Bit long field elements in projective coordinates. Hyperelliptic curves operate on divisors. In our case for $m = 81$ a divisor consists of four 81 Bit in affine and five 81 Bit field elements in projective coordinates. To handle divisors we define a struct in C, called Divisor. This struct contains 5 `gf2mShortElement`s. Analogues to elliptic curves we have to implement divisor addition and doubling. The realization of these 2 operations in software is a straight forward implementation of the steps in tables 2.1 and 2.2.

6.1.3 Implementation of the Scalar Multiplication

We use the binary method presented in Algorithm 1 to implement the scalar multiplication for ECC and HECC. The binary method is not the fastest method to implement a scalar point multiplication but it is quick to implement and the goal of this thesis is not to implement the fastest scalar point multiplication method known. The goal is rather to investigate the performance impact of substituting different parts of software by special hardware. Therefore we only implement the binary method in this thesis.
For the HECC implementation of Algorithm 1 we use divisor group operations instead of point operations and k is $2m$ Bits long instead of m Bits (see chapter 2.3.1).

6.2 Type II: Multiplier in Hardware

The Type II design option is our first step to replace software functionality from Type I with special hardware. As can be seen from Table 5.1 the field multiplication is the slowest of the field arithmetic functions. It therefore accounts for the biggest time share of the scalar multiplication. Hence we decided that the $\mathbb{F}_{2^m}$ multiplication is the first function replaced with special hardware. In the following we show the hardware architecture of Type II and what changes in the software are required to take advantage of the added special hardware.

For ECC and HECC we implemented a Type II design option with one $\mathbb{F}_{2^m}$ multiplier in hardware. In addition to that we also implemented for HECC a Type II design with two $\mathbb{F}_{2^m}$ multipliers in hardware.

6.2.1 Type II Architecture with 1 Multiplier

Figure 6.2 shows the hardware architecture of the Type II design with one multiplier on the FPGA. The design consists of the APB Slave interface, two input registers (A and B), one output register (C), and one $\mathbb{F}_{2^m}$ multiplier-block. We use this layout for the elliptic as well as for the hyperelliptic curve implementation.

The $\mathbb{F}_{2^m}$ multiplier block in Figure 6.2 consists of a state machine and LSD multiplication core. The function of the LSD is described in chapter 5.2.2. The state machine controls the operation of the multiplication core. It also signals wether the multiplication core is busy or not. When the state machine receives the enable signal it causes the multiplication core to load the contents of the two input registers and start multiplication immediately. After the appropriate number of clock cycles, which depends on the ratio between m and the digitsize D of the multiplier, the result of the multiplication

core is ready. The state machine then ensures that this result is stored into the result register C.

For the ECC implementation of Type II the width of each of the three registers used in Figure 6.2 is 167 Bit. For the HECC implementation the registers are 81 Bit. We implement four different digitsizes D of the multiplication core D=4, 8, 16 and 32. We always implement these four digitsizes for all design options in this thesis.

To use the hardware multiplier the ARM processor has to do the following. It has to

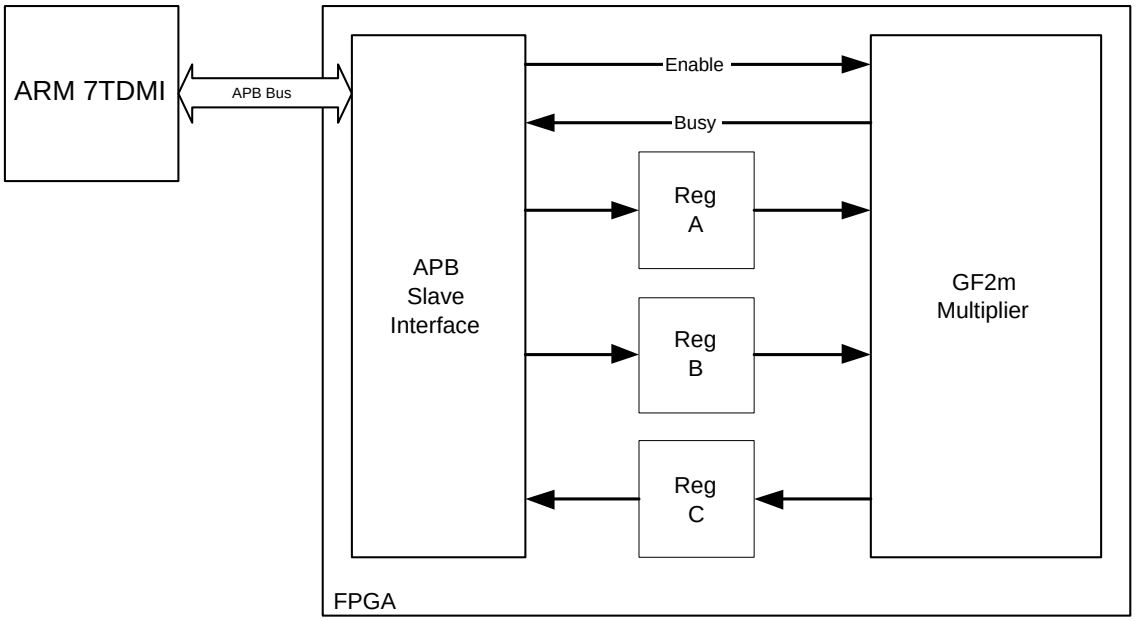


Figure 6.2: Type II: With one Multiplier in Hardware

transfer the two field elements, that are supposed to be multiplied, from its memory into the two input registers A and B on the FPGA. Then the computation on the FPGA is started. After the computation has finished the ARM has to transfer the result from register C back into its memory.

This process of sending data and picking up the result from the FPGA is handled by a rewritten version of the `gf2mMult` function on the ARM.

The width of the APB that connects the ARM and the FPGA is 32-Bit. Hence the

field elements are transferred in chunks of 32Bits to and off the FPGA. Thus it takes (m mod 32) + 1 transfers of 32Bit words to transfer one field element between the ARM processor and the FPGA, i.e. it takes 5 transfers for a 167 Bit field element and 3 transfers for a 81 Bit field element.

There are two ways of how the ARM processor can start the multiplier on the FPGA. One way is to send an extra "start multiplication" command to the FPGA. This approach implies to send an extra 32Bit word over the APB just to start the multiplier. This approach is obviously quite slow. Hence we choose a faster approach. In our implementation we let the APB slave start the multiplier automatically as soon as the last 32Bit chunk of register B has been transferred to the FPGA, thus saving the need to send an extra start command.

The FPGA is a slave of the APB, i.e. only the ARM processor can initiate transfers on the APB. There are 2 ways for the ARM to figure out the state of the hardware on the FPGA, e.g. wether the multiplier is busy or not. The active way is for the FPGA to signal a change in the state to the ARM using interrupts (IRQ). There are 29 interrupt lines connecting the FPGA to the ARM. When the FPGA sends an IRQ, then the ARM services this IRQ by leaving its normal program flow and executing the interrupt service routine that is assigned to that IRQ. In this thesis we do not use interrupts though. The other way how the ARM gets information on the state of the FPGA is to read a status flag from the FPGA via the APB. This method is called polling.

For a fast implementation of `gf2mMult` the ARM processor has to pick up the result from the FPGA right after the result is available. To do this the ARM can use the polling method just described. Polling doesn't lead to optimum results though, because it requires a loop that permanently downloads the status flag from the FPGA (always 32Bits, because of the APB structure) to the ARM. The status flag is then checked on

the ARM to figure out wether the multiplication has finished or not. If it has finished the ARM leaves the polling loop and starts transferring back the result.
There is another solution to get the result faster. This solution avoids the polling method. The number of clock cycles for a multiplication only depends on m and the digitsize D. Hence we already know in advance after how many clock cycles the computation will be done. Thus we just wait for this number of clock cycles by executing NOPs (No Operation) commands on the ARM and then start transferring the result.

6.2.2 Type II Architecture with 2 Multipliers

The hyperelliptic curve group operations use about twice as many field multiplications as the elliptic curve group operations. Most of these multiplications can be executed in parallel. Hence we implement a Type II design with two multipliers for HECC. Figure 6.3 gives an overview of the layout we use for this approach. It consists of an APB slave, two input registers (A and B), two output registers (E and F), a state machine (Control SM) and two multiplier blocks. Each of the registers is 81 Bits wide.

To take advantage of the two multipliers it is not sufficient anymore to just replace the `gf2mMult` function with a rewritten version. It is also necessary to rewrite the two group operations for divisor doubling and addition in a way that multiplications, that can be done in parallel, are grouped together in pairs. Then we define two new field multiplication functions, in addition to `gf2mMult`, to enable the parallel use of the two multipliers for these pairs. There are two cases for these pairs of multiplications. In the first case the 2 multiplications have no input field element in common. In the second case the 2 multiplications have one input field element in common. Thus in the second case we just have to transfer 3 field elements instead of 4 elements to the FPGA. We name the two new functions `gf2mMult2` for the first case and `gf2mMult3` for the pairs

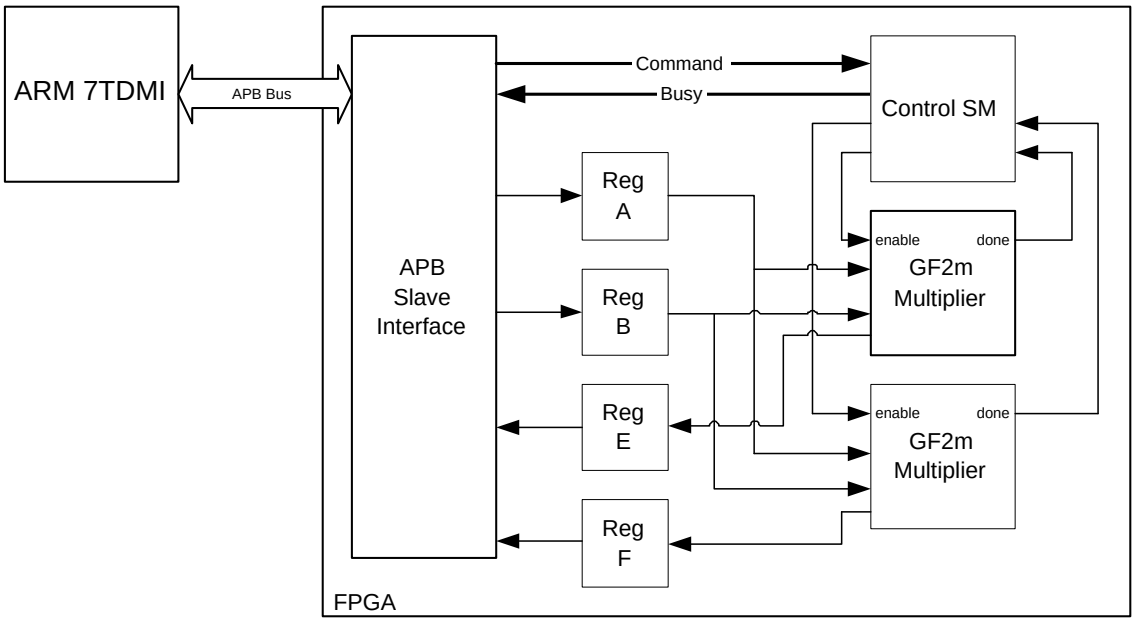


Figure 6.3: Type II: With two Multipliers in Hardware

of the second case. Of the 69 field multiplications used in the HECC group operations, 32 can be executed using `gf2mMult2` and 30 using `gf2mMult3`. This means that just 7 multiplications are left that can't be done in parallel to other multiplications.

The purpose of the state machine is to start the multipliers according to the commands sent from the APB slave and to signal wether the multipliers are busy doing a multiplication or not.

We save the cost of sending extra start commands for the multipliers from the ARM to the FPGA by using different addresses for the same physical registers on the FPGA. Let us clarify this with an example. Figure 6.4 shows how register A of this Type II architecture is blended into the memory map of the ARM processor. From the figure we see that the 32 least significant bits of register A can be accessed using address 0x60000000 on the ARM. The next 32 bits can be accessed using address 0x60000004. The last 17 bits of register A can be accessed using two different addresses, either 0x60000008 or 0x60000108. The difference is that when we access them via the second address, in

addition to just storing the data transferred via the data lines of the APB, the APB slave also starts the second multiplier. We used the same trick for register B to start the first multiplier.

The function `gf2mMult` needs no modification compared to the implementation with

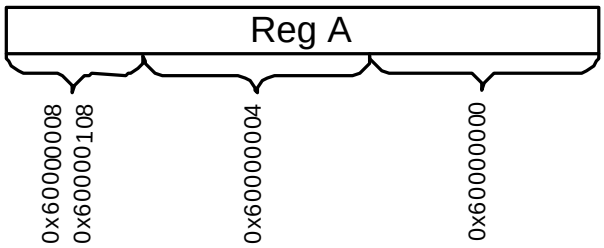


Figure 6.4: Memory mapping of register A

just one multiplier. `gf2mMult` works in the following way. The ARM transfers two field elements into the registers A and B. The first multiplier block then starts the multiplication and stores the result in register E, from where the ARM then transfers it back into its own memory. The function `gf2mMult2` (no elements in common) is identical to `gf2mMult` up to the point where the first multiplier is enabled. Then instead of waiting for the first multiplier to finish `gf2mMult2` transfers the next two field elements of the next multiplication into the registers A and B. Then the second multiplication block starts multiplication. When it has finished it stores the result in register F, from where the ARM can pick it up. This usage of the registers A and B as input for both multipliers is possible because the content of the input registers is just read by the multipliers when they start a multiplication, after that changes to the contents of these registers don't effect the outcome of the multiplication anymore. The flow for `gf2mMult3` is similar to the one of `gf2mMult2`. The only difference is that one field element does not need to be transferred again to the FPGA, because it is already stored in the FPGA from the first multiplication.

To transfer the results back to the ARM, we just wait for the appropriate number of clock cycles by issuing the right number of NOPs until the results are ready. Then we start transferring back the result. The result of the first multiplier can already be transferred back to the ARM, while the second multiplier is still computing. There is no need to wait for the second multiplier, because the APB slave handles the transfer independently from the multipliers.

6.3 Type III: AU in Hardware

The idea behind the Type III implementation is to store all intermediate values in the FPGA during one scalar multiplication. Hence eliminating the time consuming transfer of data between the ARM processor and the FPGA. The ARM processor handles the control flow by sending commands to the FPGA. Figure 6.5 shows the block diagram of the Type III design option. It consists of an APB Slave Interface, a dual ported RAM (DPRAM), a state machine, a multiplier block, an adder block and a squarer block. The block diagram for the ECC and the HECC implementation consists of the same blocks. The difference between them is that for ECC the data busses connecting the DPRAM and the field arithmetic blocks are 167 Bits wide. For HECC they are 81 Bits wide. Also the width of the field arithmetic blocks and the DPRAM are different from each other. The Type III implementation requires a lot more changes in software than Type II. The group operations for adding and doubling have to be rewritten completely. The $\mathbb{F}_{2^m}$ field arithmetic functions `gf2mMult`, `gf2mSquare` and `gf2mAdd` in software are not needed anymore because they are now entirely replaced by their hardware counterparts.

In following we will describe the different functional blocks of the Type III design.

6.3.1 The Dual Ported RAM (DPRAM)

To store all field elements that are required for a scalar multiplication on the FPGA (inclusive temporary results), we need 14 registers with a width of 167 Bits for ECC and 23 registers with a width of 81 Bits for HECC. We implement these register files in a block of dual ported RAM.

Our FPGA XCV2000E contains 160 special RAM blocks called Select-RAM+$^{\text{TM}}$. With these blocks it is possible to build true dual ported memory, which has two independent

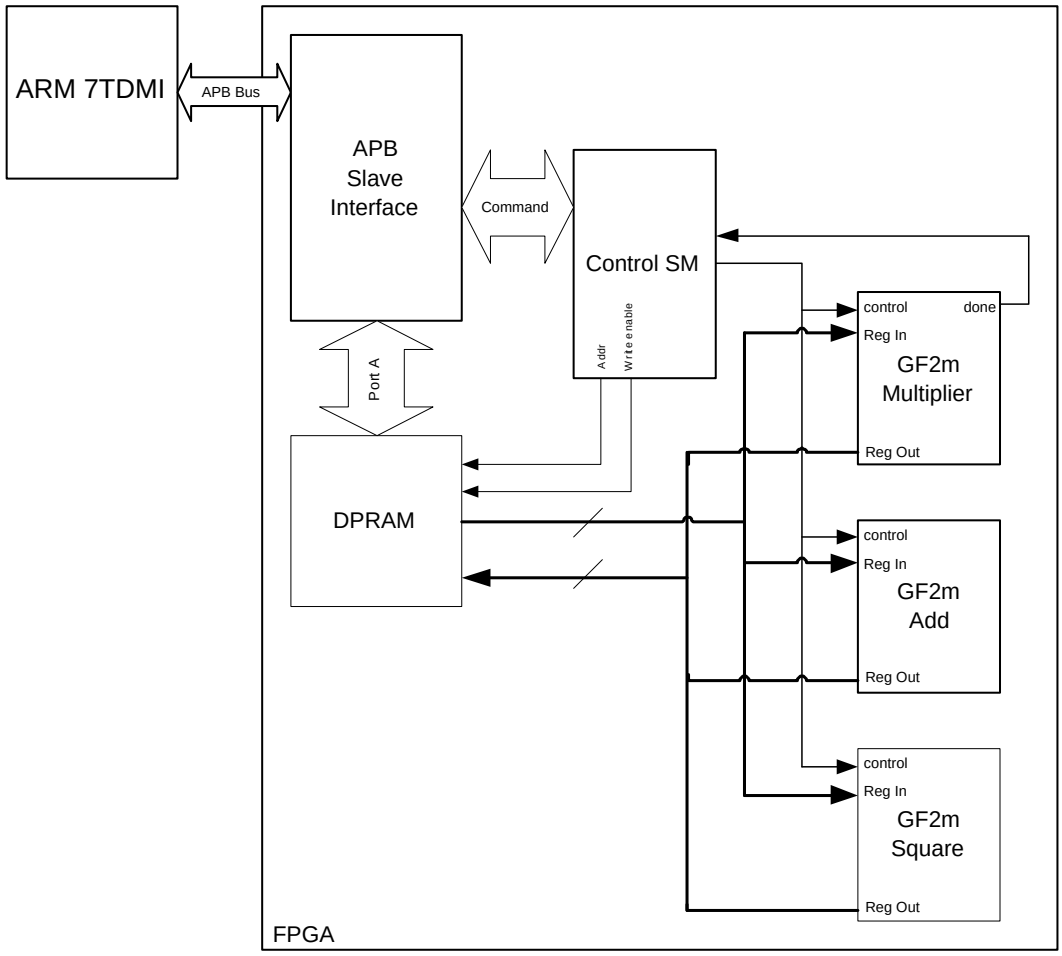


Figure 6.5: Type III: AU in Hardware with one Multiplier

data ports (A and B) to enable simultaneous access to the same physical memory. Figure 6.6 gives an overview of the DPRAM block. Each port consists of n address lines, a write enable line (Wen), m DataIn and m DataOut lines. The number of address lines depends on the number of elements that have to be addressed on that port. The address lines are used to select a memory cell. The DPRAM outputs the data from the selected memory cell within a clock cycle on the DataOut lines. The Wen line is used to write the data from the DataIn lines into the selected memory cell. The data width of port B can be 1, 2, 4, 8 or 16 times the width of port A. In our implementation port A always has a

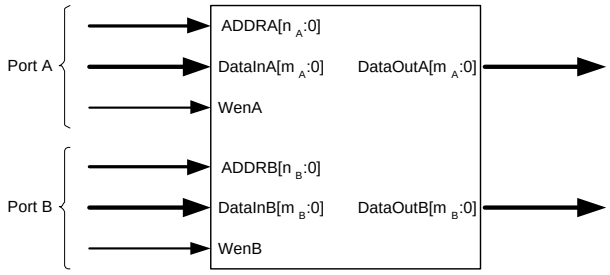


Figure 6.6: Dual Port RAM Block (DPRAM)

width of 32 Bits, i.e. $m_A = 31$ in Figure 6.6. For the ECC implementation Port B has a width of 256 Bits of which we use the lower 167 Bits on the data lines and for HECC the width of port B is 128 Bits of which we use the lower 81 Bits.

Port A of the DPRAM is directly connected to the APB Slave in order to enable the ARM processor to read from and write to the registers directly. Port B is connected to the $\mathbb{F}_{2^m}$ arithmetic units via two data busses.

One bus connects the DataOut pins of port B to the RegIn ports of the arithmetic units. This is the only bus used to transfer data to the arithmetic units. Thus whenever a unit needs two registers as input, as is the case for the multiplier and the adder, the registers have to be transferred to the units one after another. Each transfer of a field element from the RAM to the arithmetic units takes 2 clock cycles. The first clock cycle to set up the address and the second to actually store the data into the unit.

The other bus connects the DataIn pins of port B with the RegOut ports of the different arithmetic units. The different units share this bus to transfer their results back to memory. Hence we have to make sure that only one unit outputs data on the bus at a time. The other units have to be disconnected from the bus, while such a transfer takes place. The disconnection is done by putting the RegOut of the different units into a high impedance state. The access to the bus is controlled by the control state machine (Control SM). The control state machine grants access to the bus by sending a signal

to the arithmetic unit that is supposed to output its data onto the bus. Only while this enable signal is set does the selected arithmetic unit output its result on the bus. If the signal is not set anymore the arithmetic unit disconnects itself from the bus. A write access to the RAM takes 2 clock cycles, one to set up the address and the other one to store the data into the RAM.
The address lines and the write enable line (Wen) of the DPRAM are connected to the control state machine. The address lines are used to select a register in the RAM for a read or write access. The write enable line is used to write the data on the DataIn lines of the RAM into the selected register.

6.3.2 The Multiplier Block

The Multiplier block consists of a LSD multiplication core, which is described in chapter 5.2.2 with $m = 167$ for ECC and $m = 81$ for HECC. In addition to the multiplication core the block also consists of a small state machine, 2 input registers, and one register for results. The state machine controls the operation of the multiplication core. When the state machine receives an enable signal from the control state machine then it loads the two input registers into the multiplication core. It stores the result of the multiplication in the result register when the multiplication has finished and sends a done signal to the control state machine to signal that the result is available.

6.3.3 The Squarer Block

The squarer block basically consists of one input register and one output register. The bits of the input register are connected to the output register as described in chapter 5.2.3.

6.3.4 The Adder Block

The adder block consists of 2 input registers and one result register. The contents of the result register are the result of a bitwise XOR of the two input registers.

6.3.5 The Control State Machine

The Control state machine (control SM) receives commands from the ARM processor and coordinates the function of the arithmetic units and of port B of the DPRAM accordingly. The state machine can interpret three different commands, one for each arithmetic unit. The commands multiply and add have two source and one target register, square has one source and one target register. The source and target registers can be any of the registers stored in the DPRAM.
The control SM only accepts new commands from the ARM, when the execution of the previous command has finished. When the state machine accepts a new command it setups the address lines of the DPRAM according to the first source register and then loads the data into the first input register of the target arithmetic unit. For multiply and add it then setups the memory address according to the second source register of the command and then loads that data into the second input register of the according unit. In the case of a multiply it then enables the multiplier and waits for the multiplier to signal that the multiplication has finished. The last step for all three commands is to store the result into the DPRAM with the address from the target register of the command. Let us clarify this with an example. When the ARM sends a `fadd(r3,r1,r2)` to the control state machine then the state machine loads register r1 from the DPRAM into the first input register of the adder block. Then it loads register r2 from the DPRAM into the second input register of the adder block. Finally the state machine stores the

result in register r3 of the DPRAM.

6.3.6 Changes in the Software on the ARM

We implement the three field arithmetic commands, multiply, add, and square, in C as macros. The macros enable us to use the different commands in a quite convenient (assembler like) style. We created the following three macros:

```
fmul(r3,r1,r2)
```

```
fadd(r3,r1,r2)
```

```
fsqr(r3,r1)
```

In these macros r3, r1 and r2 can be any of the available registers on the FPGA, with r3 being the target register and r1 and r2 being the source registers. In the ECC implementation the FPGA has 14 registers named with hex numbers from $0x0$ to $0xd$ and for HECC the FPGA has 23 registers named from $0x00$ to $0x17$.

The group operations have to be rewritten completely to take advantage of the special hardware. Instead of calling the software functions for field arithmetic the rewritten operations now use the corresponding macros. The main loop for the scalar multiplication stays the same, but instead of calling the software functions we now call the rewritten group operations. Before we start the scalar multiplication the registers in the DPRAM of the FPGA have to be set up. To do this we have to transfer P to the appropriate registers and we have to zero the registers that will store the result Q of the scalar multiplication. After the scalar multiplication has finished we have to transfer the result back to the ARM.

6.3.7 Type III implementation with 2 multipliers for HECC

We also implement a Type III design with 2 multipliers in hardware for HECC (Figure 6.7). We do this for the following reasons:

- HECC uses 69 field multiplications in its group operations
- 62 of these multiplications can be executed in parallel pairs

The only difference in the block diagram of this option is the additional multiplier as can be seen when comparing Figures 6.7 and 6.5. To take advantage of the additional multiplier the control state machine has to be modified. We include the following two new commands in the state machine:

`fmul2(t1,t2,in11,in12,in21,in22)`

`fmul3(t1,t2,in11,in3,in22)`

In these functions the register t1 is the target register of the first multiplier and t2 the target register of the second multiplier. Register in11 and in12 are the source registers of the first multiplication and registers in21 and in22 are the source registers of the second multiplication. Register in3 of `fmul3` is the shared input register to the first and the second multiplication.

Only the two group operations for doubling and addition have to be rewritten using the 2 new commands on the ARM.

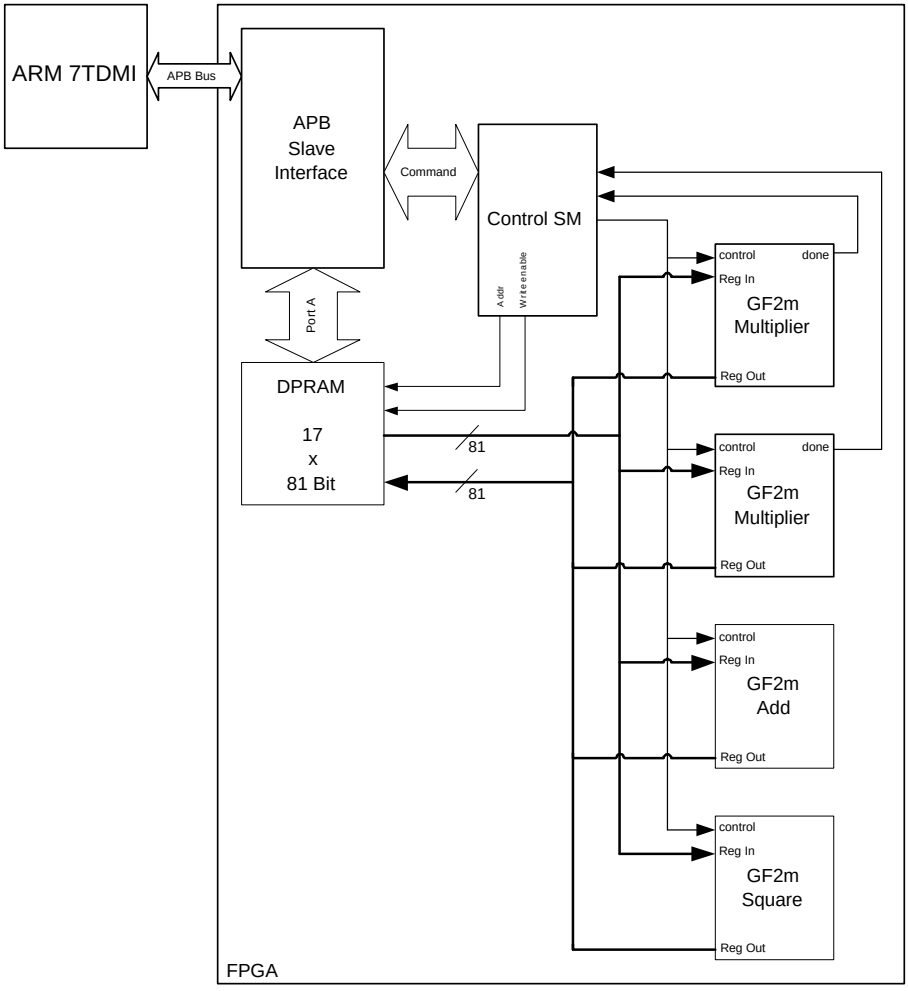


Figure 6.7: Type III: HECC implementation with 2 multipliers

6.4 Type IV: Hardware only

To further increase the performance we implement the complete scalar multiplication in the FPGA. In the design option Type IV the ARM processor is only used to transfer the scalar k and the point (or divisor) P to the FPGA. Then the processor sends a run command to the FPGA and waits for the scalar multiplication to finish. After that the ARM transfers the result of the multiplication back from the FPGA to its own memory. Figure 6.8 provides an overview of the architecture of the ECC implementation. Figure 6.9 shows the HECC block diagram of Type IV. The difference between the two figures is mainly the connection of the shifter to the APB slave in the HECC architecture.

6.4.1 Overview Functional Blocks

The three field arithmetic units are the same units used in the Type III design. Hence we do not cover them in this section again. The biggest changes compared with Type III are the addition of a shifter, a main state machine and a program memory. In the following we will describe these new blocks and their functionality.

The Shifter

To implement the binary method for scalar multiplication in hardware we need to determine wether bit k_i of the scalar k is set or not. To solve this problem we added a shifter to our design. For ECC we added a 167 Bit shifter that is connected to the DataOut of the DPRAM like the other arithmetic units. It doesn't have a connection to the DataIn of the DPRAM, because there is no need for the shifter to write data back into the memory.

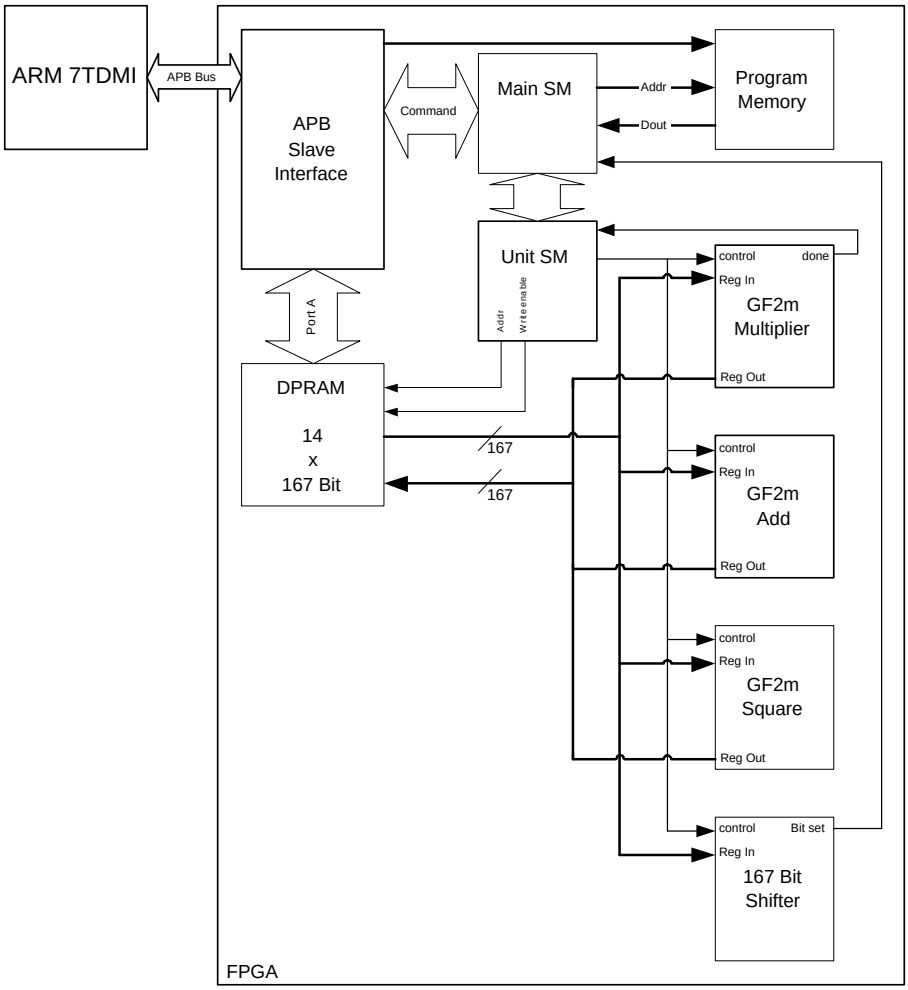


Figure 6.8: Type IV: Hardware only (ECC implementation)

We implement the scalar multiplication using the left-to-right binary method. Hence we implement the shifter as a left shifter. In the ECC implementation the shifter consists of a 167 Bit wide register and some logic to shift the data stored in that register by one bit whenever a shift signal is received.

Figure 6.10 shows how the shifter works. The most significant bit k_m stored in the shifter is shifted into the least significant bit k_0, bit k_0 is shifted into k_1 and so on.

The shifter outputs a signal "BitSet". This signal outputs the content of bit k_0. If for example we want to know if bit k_i is set, we simply shift it into bit k_0 and then check

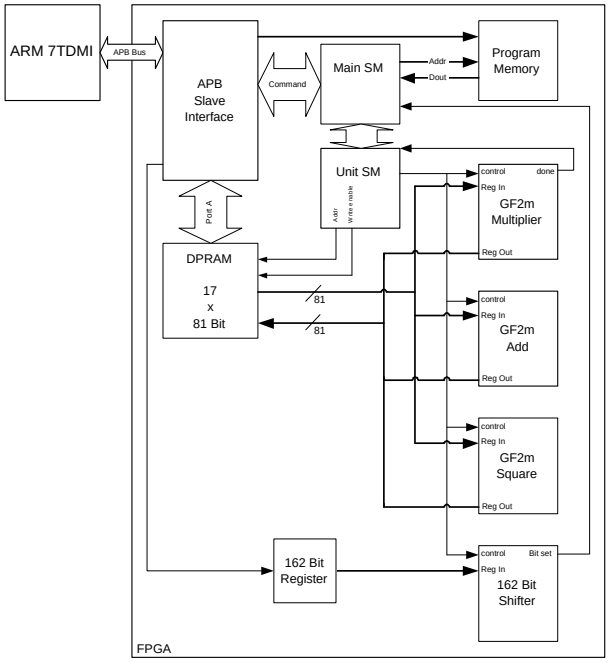


Figure 6.9: Type IV: Hardware only (HECC implementation)

the BitSet signal.

The HECC shifter works the same way as the ECC shifter. Because in HECC k is $2m$ Bits long the shifter's register in the HECC implementation is 162 Bits wide.

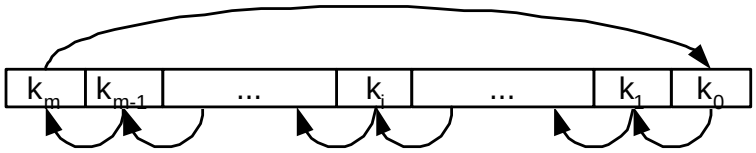


Figure 6.10: Type IV: Shifter

The program memory

The program memory holds the complete code sequence to do a scalar multiplication. In the ECC implementation the program memory can store programs with a length of 128 instructions. In HECC it can store up to 256 instructions. The program memory is

implemented as dual port memory with one port connected to the APB slave to enable the ARM processor to transfer program code into the memory. The program upload happens when the system starts up. After the program code has been transferred from the ARM to the FPGA it stays in the program memory until the FPGA is being reset or the code is overwritten with new code by the ARM.
The other port of the program memory is connected to the main state machine and is used during program execution.

The state machines

The unit state machine (Unit SM) in Figures 6.8 and 6.9 has the same abilities as the control state machine from the Type III design. In addition to that it also sends control signals to the shifter. The unit state machine receives the commands it has to execute from the main state machine and it signals back to the main state machine when it has finished executing a command.
The main state machine is responsible for actually executing programs that are stored in the program memory. To run programs the main state machine features a program counter (PC) that points to an address in the program memory. During program execution the PC is incremented after each instruction to point to the next instruction, which is then executed. To enable loops in the program flow we implement conditional and unconditional jump instructions. These jump instructions can set the PC to any address of the program memory.
To start program execution the ARM processor can set the program counter to any value. After the PC has been set program execution automatically starts from that address.
The main state machine also contains a counter. The counter can be set to any value

within programs. There is also a command to decrement the counter. The counter is required to implement loops like the main loop of the scalar multiplication.

6.4.2 Instruction set of the type IV design

Our Type IV design supports the following instructions to implement the scalar multiplication on the FPGA:

`mul(t,s1,s2)` performs a $\mathbb{F}_{2^m}$ multiplication of the two field elements stored in the registers s1 and s2 and stores the result in register t.

`add(t,s1,s2)` performs a $\mathbb{F}_{2^m}$ addition of the two field elements stored in the registers s1 and s2 and stores the result in register t.

`sqr(t,s1)` performs a $\mathbb{F}_{2^m}$ squaring of the field element stored in register s1 and stores the result in register t.

`moveReg(t,s1)` copies the field element stored in register s1 to the register t.

`setZero(t)` sets the content of register t to zero.

`ldshifter(s1)`. In the ECC implementation this instruction loads the shifter with the value stored register s1. In the HECC implementation just the value from the register connected to the shifter is loaded (see Figure 6.9).

`shift` causes the shifter to shift its content left by one bit.

`setCounter(value)` loads the counter within the main SM with `value`.

`decCounter` decrements the counter by one.

`Jmp(addr)` is an unconditional jump and sets the PC to addr.

`JmpKbitSet(addr)` is a conditional jump. It sets the PC to `addr` when the BitSet output of the shifter is set. Otherwise the PC is just incremented by one.

`JmpCounterZero(addr)` is a conditional jump. It sets the PC to `addr` when the Counter of the main state machine is zero. Otherwise the PC is just incremented by one.

`End` is used to stop program execution. This instruction makes the main SM leave its program execution cycle to return into the idle state.

6.4.3 Changes in the Software on the ARM

The ARM processor has to write the program code for the scalar multiplication into the program memory of the FPGA. The length of the ECC scalar multiplication program is 54 instructions. The HECC program is 155 instructions long. This programming of the FPGA has to take place before the scalar multiplication on the FPGA is executed for the first time. After that no reprogramming is required, until the FPGA is being reset.

Once the scalar multiplication program has been uploaded into the program memory of the FPGA, the ARM just needs to do the following to execute a scalar multiplication on the FPGA. The first step is to transfer the scalar k and the point (or divisor for HECC) P to the FPGA. Then the ARM starts the multiplication on the FPGA by setting the program counter to the beginning of the multiplication program. After the multiplication has finished, that is when the main state machine has returned to idle state, the ARM transfers the result back from the FPGA into its own memory.

7 Results

In this chapter we present and discuss the results we obtained for the different types of ECC and HECC implementations. It is important to point out that all time measurements are done with the ARM processor and the FPGA running at a clock frequency of 25 MHz.

As a general result from our work we can say that ECC and HECC can be accelerated considerably by using specialized hardware. Our fastest elliptic curve scalar multiplication using special hardware takes 1.9 ms. This is 390.4 times faster than the software only version we started with. The fastest hyperelliptic scalar multiplication takes 6.2 ms, which is 248.4 times faster than the software only implementation. In the following we will first discuss the results obtained from the ECC implementations. Then we will discuss the results from the HECC implementations. After that we will compare the ECC with the HECC implementations.

7.1 Presentation of ECC Results

The results we obtained for the different ECC implementations are shown in the Tables 7.1, 7.2, 7.3, and 7.4. Let us first describe these tables, before we discuss the results at the end of this section.

Table 7.1 lists the cost of hardware and the performance of the different ECC im-

Design Option	Digit-size	Codesize [kB]	Slices	Time for k∗P [ms]	Speedup compared to Type I
Type I: SW only		33.53	0	741.8	1.0
Type II: Multiplier in HW	4	25.35	816	53.9	13.7
	8	25.33	1170	51.2	14.5
	16	25.32	1563	50.7	14.6
	32	25.32	2799	50.3	14.7
Type III: AU in HW	4	24.67	1053	5.3	140.0
	8	24.38	1400	4.3	172.5
	16	24.20	1793	3.5	211.9
	32	24.15	3020	3.2	231.8
Type IV: HW only	4	23.59	1394	4.2	176.6
	8	23.59	1648	2.9	255.8
	16	23.59	2141	2.2	337.2
	32	23.59	3220	1.9	390.4

Table 7.1: ECC Cost and Performance of Implementations

plementations. The first column shows the design option. The second one indicates the digitsize of the $\mathbb{F}_{2^m}$ multiplier used in that particular implementation. In the third column we list the code size of the executable image for the ARM processor. Besides the routines actually required for ECC the code size also includes the space needed for all other routines compiled into the image, e.g. board initialization and serial port I/O routines. The fourth column shows the number of slices the design actually occupies on the FPGA after the place&route step of the hardware compiler. The column 'Time for k*P' shows the time one scalar multiplication takes on average in milliseconds, when the result is presented in standard projective coordinates. To transfer the result to standard affine coordinates one additional inversion and two field multiplications are required. The timing listed in Table 7.1 does not include the time for these three additional operations. It is important to mention that we did not implement the field inversion in hardware. According to [51], who also used an ARM 7TDMI in his thesis, a $\mathbb{F}_{2^m}$ field

inversion with $m = 167$ takes about 8 times longer when implemented on this processor than a field multiplication. I.e. with the software timings from Table 7.4 it takes about 4.6 ms on the ARM to convert the result from projective to affine coordinates.
For Type III and IV the column 'Time for k*P' includes the time all the necessary data transfers require. This includes transferring the input data for the scalar multiplication to the FPGA and it also includes transferring the result back into the memory of the ARM processor. The last column of Table 7.1 shows the relative speedup of each implementation compared to Type I.

Table 7.2 shows in detail how many resources are actually occupied on the FPGA for

Design Option	Digit-size	Slices	Slice Flip Flops	4-input LUT	Max. Frequency [MHz]	BlockRam Cells
Type II: Multiplier in HW	4	816	1086	1085	120.6	0
	8	1170	1125	1541	121.1	0
	16	1536	1206	2407	133.9	0
	32	2799	1531	4431	113.4	0
Type III: AU in HW	4	1053	1649	1224	78.6	16
	8	1400	1687	1679	78.6	16
	16	1793	1767	2539	78.6	16
	32	3020	2094	4557	78.6	16
Type IV: HW only	4	1394	1986	1907	78.6	17
	8	1684	2045	2427	78.6	17
	16	2141	2124	3281	78.6	17
	32	3220	2282	5305	78.6	17

Table 7.2: ECC Hardware Costs in Detail

each design. The first three columns are the same ones as in Table 7.1. The next two columns show the number of flip flops and of 4-input Look Up Tables used. The maximum frequency column shows at which the FPGA could be clocked for this particular design implementation. The last column shows how many block RAM cells are used in each design.

The Table 7.3 presents the execution times of the two group operations double and

Design Option	Digit-size	Point Double [μs]	Point Add [μs]
Type I: SW only		2240	4500
Type II: Multiplier in HW	4	228	332
	8	206	325
	16	203	320
	32	202	318
Type III: AU in HW	4	17	29
	8	13	21
	16	10	16
	32	10	15
Type IV: HW only	4	15	26
	8	10	18
	16	8	14
	32	7	12

Table 7.3: ECC Timings Group Operations

add for the different implementations in μs. The time it takes to transfer the results of the group operations back into the memory of the ARM processor is not included for Type III and IV.

Table 7.4 lists the number of clock cycles it takes to execute the different field arithmetic functions. For the $\mathbb{F}_{2^m}$ multiplication the table also shows how the number of clock cycles is composed. For the Type II multiplication this number is composed of three parts, which add up to the total number of clock cycles. For example the Type II with $D = 4$ (Table 7.4, 3^{rd}row, 3^{rd}column) field multiplication takes $42 + 2 + 300 = 344$ clock cycles. This means that the 344 clock cycles are composed of 42 cycles for the field multiplication, 2 cycles to load the input values from the registers and to store the result back into the result register. The 300 clock cycles are needed for the communication between the ARM and the FPGA.

The numbers of clock cycles shown for Type III and IV are composed differently. For

Design Option	Digit-size	gf2mMul [clock cycles]	gf2mSquare [clock cycles]	gf2mAdd [clock cycles]
Type I: Sw only		11480	836	169
Type II: Multiplier in HW	4	42+2+300=344	344	169
	8	21+2+300=323	323	169
	16	11+2+300=313	313	169
	32	6+2+300=308	308	169
Type III: AU in HW	4	42+6=48	4	6
	8	21+6=27	4	6
	16	11+6=17	4	6
	32	6+6=12	4	6
Type IV: HW only	4	42+6=48	4	6
	8	21+6=27	4	6
	16	11+6=17	4	6
	32	6+6=12	4	6

Table 7.4: ECC Timings Field Arithmetic

example the Type III multiplication with $D = 8$ (Table 7.4, 8^{th}row, 3^{rd}column) needs 21+6 = 27 clock cycles, 21 cycles for the multiplication and 6 cycles load 2 field elements from the DPRAM into the multiplier block and to store the resulting field element back into the DPRAM.

7.2 Discussion of ECC Results

Here we discuss the different results shown in Tables 7.1, 7.2, 7.3, and 7.4. We discuss the results beginning with the field arithmetic functions from Table 7.4.

In Table 7.4 one notices that the Type II design timings for gf2mSquare are identical to the timings for gf2mMult. That is the case because even the slowest Type II gf2mMult (Table 7.4, 3^{rd}row, 3^{rd}column)is still faster than the software implementation of gf2mSquare which needs 836 clock cycles. Hence we implemented gf2mSquare using the Type II gf2mMult function.

We did not implement a special hardware squarer. In a Type II setup a special squarer in hardware would need $2 + 200 = 202$ clock cycles (2 clock cycles for the squaring and 200 clock cycles to transport data between the ARM and the FPGA), which is faster than the fastest Type II gf2Mult from Table 7.4. The fastest gf2mMult uses 308 clock cycles (Table 7.4, 6^{th}row, 3^{rd}column). Thus using such a special hardware squarer would result in a further increase of performance of the Type II squaring. But if we look at the performance gain of a scalar multiplication when using such a hardware squarer, the gain is too small to justify the extra hardware cost. Hence we did not implement a hardware squarer in a Type II like setup.

In Table 7.4 the number of clock cycles for gf2mAdd are identical for Type I and II. A special hardware adder in a type II setup would need $2 + 300$ clock cycles to add up two field elements, which would be much slower than the 169 clock cycles of the software implementation on the ARM processor, hence we use the Type I gf2mAdd function for Type II as well.

For Type III and IV it takes 2 clock cycles to load one field element from the DPRAM into one of the arithmetic units. Storing one field element into the DPRAM also takes 2 clock cycles. The actual $\mathbb{F}_{2^m}$ addition and squaring in the arithmetic units is done instantly. Thus a Type III or IV squaring always takes 4 clock cycles (Table 7.4, 4^{th}column), 2 cycles to load an field element from the RAM and 2 cycles to store the result into the RAM. The Type III or IV addition takes 6 clock cycles, 4 cycles to load two field elements from the RAM and 2 cycles to store the result back to the RAM.

In Table 7.1 we see that increasing the digitsize does not lead to a considerable gain in performance for the Type II designs, though the amount of hardware used on the FPGA grows with every increase of the digitsize as we see from Table 7.1 and 7.2. Compared to the software only implementation the speedup factor for $D = 4$ is 13.7. For $D = 8$

the speedup is 14.5. To achieve this performance gain 1170 slices instead of 816 are used on the FPGA. This is an increase in hardware complexity of 43%. For $D = 16$ the speedup is 14.6 with 1536 slices used on the FPGA, which is an growth in hardware complexity of 34% compared to $D = 8$. For $D = 32$ the speedup is only 14.7 with an hardware increase of 79% compared to $D = 16$. This enormous increase in hardware cost stands in no relationship to the achievable performance gain anymore. The reason for this behavior lies within the concept of the Type II design. While the actual multiplication becomes twice as fast for every doubling of the digitsize, the time to transfer data between the ARM and the FPGA stays the same (see table 7.4, 3^{rd}column, rows 3 - 6). The transport always takes 300 clock cycles. This is the time it takes to transfer two 167 Bit field elements from the ARM's memory to the FPGA and the resulting field element back. For a digitsize of $D = 4$ the transfer time accounts for 87% of the execution time of `gf2mMult`. For $D = 32$ the transportation already accounts for 97% of the execution time. Hence the transfer between the ARM and the FPGA is the dominating bottleneck for design Type II.

As a result of this behavior we can say that only small digitsizes make sense for the Type II design. The small gains in performance for higher digitsizes do not weigh up the added hardware complexity.

To increase the execution speed further we have to remove the data transportation bottleneck between the ARM and the FPGA. One way to do this is to accelerate the communication between the ARM processor and the FPGA and thereby widening up the transportation bottleneck. The FPGA is connected to the ARM via the Advanced Peripheral Bus (APB) (see Figure 4.3). The APB is intended to be used for low bandwidth peripherals. A connection between the processor and the FPGA via the faster AHB would be a much better solution for our needs. But because we cannot change

the connection between the FPGA and the ARM, we have to find another way to work around the bottleneck.

This leads us to the design Types III and IV. These two types hold all the data they need for one scalar multiplication within a memory on the FPGA. Hence the amount of data transferred between the ARM and the FPGA is kept to a minimum. For design Type III the communication only consists of commands being send, which requires only a fraction of the bandwidth that the transportation of field elements in Type II requires. For design Type IV the necessary data transport is reduced even more compared to Type III. Type IV requires no more data transport between the ARM and the FPGA once a scalar multiplication has been started. This reduction in required bandwidth between the ARM and the FPGA effectively works around the transportation bottleneck. From the 6^{th}column of Table 7.1 we see that with the removal of Type II's bottleneck the speedup factor makes a big jump when switching to Type III and IV. For example for Type III with $D = 4$ (Table 7.1, 7^{th}row, 6^{th}column) the speedup factor is 140, which is almost 10 times faster then the fastest Type II design. This speedup is achieved with only 1053 slices used. The reason for this speedup jump lies not only within the saved transportation time. With the transport latency removed it becomes feasible to implement a squarer and an adder in hardware, which wouldn't have made sense for the Type II design. The hardware adder of Type III and IV for example just takes 6 clock cycles, which is 28 times faster than the software implementation used in Type I and II. The hardware squarer needs 4 clock cycles, which is between 77 (for $D = 32$) and 86 (for $D = 4$) times faster than the Type II square.

The speedup jump of Type III is achieved by adding additional hardware. The extra amount of hardware being added when switching from Type II to Type III is when comparing the number of slices used not very much. The two new field arithmetic blocks

(add and square) and the extra state machine in Type III add about 230 slices to the design. This can seen from Table 7.1 when comparing Type II and III designs with the same digitsize. It is important to note that the additional memory that we added to the FPGA to remove the transportation bottleneck is not included in the slice count. The added RAM can only be seen when looking at the last column of Table 7.2.

To hold all the data we need for one scalar multiplication we need 14 registers with a width of 167-Bit each on the FPGA. Hence the size of the RAM we use on the FPGA is 2338 Bits, which is less than 293 Bytes. The actual number of block RAM cells used for Type III in Table 7.2 is 16. Each block RAM cell can store up to 4096 Bits. Thus the amount of memory that our Type III design actually occupies on the FPGA has 8192 Bytes, which is a lot more than we actually use. But if we would implement our Type III design on an ASIC instead of the FPGA we would just need to implement 2338 Bits of memory.

Every instruction in our Type IV ECC implementation occupies 16 Bits. Hence the program memory of the Type IV design occupies an additional 128*16 Bits = 2048 Bits. The program for the ECC scalar multiplication is only 54 instructions long, thus a program memory with a size of 54*16 Bits = 864 Bits would be sufficient.

With the transportation bottleneck between the ARM and the FPGA removed, the performance of the type III design scales much better with the digitsize (see Table 7.1). The performance of Type III is limited by new bottleneck however. The new bottleneck is the access time to the DPRAM, which becomes more and more important the higher the digitsize gets. Hence the relative increase in performance gets smaller for high digitsizes. This bottleneck of the Type III design is also the bottleneck of the Type IV design. As we can see from Table 7.4 it takes 6 clock cycles to read two registers and write the result back to one register for Type III and IV. For the Type III with $D = 4$ the access time

to the memory accounts for 12.5% of the total multiplication time (Table 7.4, 7^{th}row, 3^{rd}column). For $D = 8$ the share rises to 22%, for $D = 16$ to 35% and for $D = 32$ it already accounts for 50% of the multiplication time. Hence the performance gain is much lower when switching from $D = 16$ to $D = 32$ as for example when switching from $D = 4$ to $D = 8$.

Another reason for the slower increase in performance for high digitsizes is that the time for adding and squaring stays the same for all digitsizes (see columns 4 and 5 of Table 7.1). For higher digitsizes the share of time that $\mathbb{F}_{2^m}$ add and square contribute to the total time of one scalar multiplication rises.

The performance gain when changing from Type III to Type IV design is rather small. The Type IV design performs better for higher digitsizes than Type III though. The speedup factor reaches from 176.6 for $D = 4$ up to 390.4 for $D = 32$.

The better performance of type IV comes mainly from the time saved that the ARM processor wastes in the Type III design, while waiting for a command to finish in order to send the next command. During this process there are always a couple of clock cycles wasted before the next command is actually send to and executed on the FPGA. This waste is removed completely by the Type IV design. Another reason for the performance gain over Type III is the included shifter in design Type IV.

The additional hardware effort of Type IV compared to Type III lies between 200 slices for $D = 32$ and 348 slices for $D = 16$. These additional slices are mainly added by the shifter and the main state machine.

The maximum frequencies at which the FPGA can be clocked vary from 78.6 MHz for Type III and IV up to 133.9 MHz for the fastest type II implementation (see Table 7.2). Especially the type IV design would benefit from a higher clock frequency. If the type IV design would be clocked at 75 MHz then this would result in an additional speedup

factor of three. This would effectively cut down the execution time for a scalar multiplication of type IV designs to one third of the timings listed in Table 7.1.
The maximum frequency for type II is mainly influenced by the maximum frequency at which the multiplier block can be clocked. For type III and IV the main limiting factor are the RAM cells for the registers. These block RAM cells are scattered around the FPGA, which leads to long path delays and this reduces the maximum frequency. The increased hardware hardware complexity is also responsible for the reduction of the maximum clock frequency. In general we can say that the more complex the hardware gets the lower does the maximum clock frequency sink.

Before we present the results of HECC, some words on where to use the different design Types presented in this thesis. Design Type I has the lowest hardware complexity of all the design types presented, but it is also the slowest design option. It is suitable for applications, where the emphasis is on low cost of hardware and where the speed of ECC applications is not important. Design Type II is suitable for applications where still the cost in hardware has to be kept low, but the performance regarding ECC is more important. Type III and IV cost in terms of hardware considerably more than Type II, due to their usage of RAM. The Type III is suited for embedded applications where the ECC performance is very important, but the cost of hardware is still of concern. Therefore a Type III would most probably be implemented with low digitsizes to keep the cost of hardware as low as possible. Finally Type IV is suited for systems where the emphasis lies only on performance for ECC. In such a scenario the cost of hardware is not the limiting factor anymore, hence Type IV would probably be implemented with high digitsizes.

7.3 Discussion of HECC Results

The columns of the HECC tables are basically the same ones as the ECC tables, thus we just refer to the explanations in chapter 7.1 for an explanation of the different columns. The Tables 7.5, 7.6, 7.7, and 7.8 present the HECC results of our implementations.

All of the ECC observations also apply for HECC. The following list summarizes some

Design Option	Digit-size	Codesize [kB]	Slices	time for k∗P [ms]	Speedup Factor compared to Type I
Type I: SW only		28.96	0	1539.9	1.0
Type II: 1 Multiplier in HW	4	26.08	416	94.9	16.2
	8	26.07	581	92.0	16.7
	16	26.07	775	90.6	17.0
	32	26.06	1387	89.1	17.3
Type II: 2 Multiplier in HW	4	26.23	718	86.8	17.7
	8	26.22	1048	85.7	18.0
	16	26.21	1444	85.3	18.1
	32	26.21	2651	85.0	18.1
Type III: AU in HW (1 Multiplier)	4	25.32	538	15.5	99.3
	8	24.52	708	11.8	130.5
	16	24.25	902	10.6	145.3
	32	23.98	1505	9.4	163.8
Type III: AU in HW (2 Multiplier)	4	24.45	974	11.4	135.1
	8	24.07	1311	9.7	158.8
	16	23.83	1678	8.9	173.0
	32	23.75	2892	8.2	187.8
Type IV: HW only	4	22.38	884	11.7	131.6
	8	22.38	1028	8.6	179.1
	16	22.38	1261	7.1	216.9
	32	22.38	1794	6.2	248.4

Table 7.5: HECC Cost and Performance of Implementations

results and observations that are true for ECC and HECC:

- in Table 7.8 the type II `gf2mSquare` and `gf2mMult` have the same number of clock cycles, because squaring is implemented using the `gf2mMult` routine, which is faster

as the software implementation of square on the ARM

- the Type II `gf2mAdd` is the same routine as the routine used in the Type I design, because a special hardware adder would actually be slower than the software implementation in a Type II like design option
- the data transportation bandwidth between the ARM and the FPGA is the limiting factor (bottleneck) of the Type II design
- the Type II's performance does not increase much for high digitsizes due to the bottleneck
- the Type III and IV designs remove the bottleneck of Type II by adding extra memory to store all needed data for a scalar multiplication locally in the FPGA
- the bottleneck of the Type III and IV designs is the access time to the DPRAM
- the performance of the Type IV design scales better with the digitsize than the performance of the Type III

The timings in Table 7.5 for the scalar multiplication are based on applications using only projective coordinates. To transfer the results to standard affine coordinates we would need to do one extra field inversion and 4 field multiplications. According to [51] a field inversion over $\mathbb{F}_{2^m}$ with $m = 81$ is 6.2 times slower than a field multiplication implemented in software on the ARM 7TDMI. I.e. a field inversion for $m = 81$ on our target platform takes about 1.2 ms, thus converting the result to affine coordinates takes about 2 ms.

For HECC we implemented Type II and III with two different options. One option with one and the other option with two $\mathbb{F}_{2^m}$ multipliers in hardware. Our main focus in this

Design Option	Digit-size	Slices	Slice Flip Flops	4-input LUT	Max. Frequency [MHz]	BlockRam Cells
Type II: 1 Multiplier in HW	4	416	537	553	121.4	0
	8	581	561	804	137.4	0
	16	775	610	1284	134.4	0
	32	1387	786	2236	115.4	0
Type II: 2 Multiplier in HW	4	718	907	1091	121.4	0
	8	1048	955	1633	137.4	0
	16	1444	1053	2554	134.4	0
	32	2651	1404	4443	115.4	0
Type III: AU in HW (1 Multiplier)	4	538	834	629	121.4	8
	8	708	862	900	137.4	8
	16	902	910	1359	134.4	8
	32	1505	1080	2308	115.4	8
Type III: AU in HW (2 Multiplier)	4	974	1471	1126	137.4	8
	8	1311	1518	1665	135.3	8
	16	1678	1606	2590	134.4	8
	32	2892	1958	4489	115.4	8
Type IV: HW only	4	884	1256	1328	85.1	10
	8	1028	1297	1580	84.8	10
	16	1261	1344	2006	84.8	10
	32	1794	1445	3002	84.8	10

Table 7.6: HECC Hardware Costs in Detail

section is to compare these two options for each design type.

The motivation for the implementation of these two multiplier versions is to take advantage of the possible execution of $\mathbb{F}_{2^m}$ multiplications in parallel for HECC (see also Chapter 6.2.2). As can be seen from table 7.5 the speedup factors for the Type II implementation with one multiplier reach from 16.2 for $D = 4$ to 17.7 for $D = 32$. For the implementation with two multipliers the factors reach from 17.7 to 18.1 depending on the digitsize. It is interesting to point out that even the slowest two multiplier version with $D = 4$ is faster than the fastest one multiplier version with $D = 32$, even though the one multiplier version uses with 1387 slices almost twice as much hardware as the two multiplier version with its 718 slices. As can be seen from from Table 7.5 the

hardware effort to implement the two multiplier version is almost twice as high as the effort of implementing a one multiplier version of the same digitsize. The speed gain of the two multiplier option over the option with one multiplier is mainly the result of a more efficient use of the available transportation bandwidth between the ARM and the FPGA. One reason for the extra efficiency is that the transport of input data for the second multiplier is done while multiplier one is computing its result. After multiplier one has finished the result can already be picked up, while multiplier two is still running. Another reason for the extra efficiency is that `gf2mMult3` saves the transfer of one field element, because one field element is used twice as input for the first and the second multiplier and therefore just three instead of four 81 Bit elements have to be transferred. The relative performance gain of the two multiplier version becomes negligible small for higher digitsizes, for example when changing from $D = 16$ to $D = 32$. Hence the extra hardware for high digitsizes is not worth it. The reason for this behavior is again the already mentioned transportation bottleneck of the Type II design. To accelerate the scalar multiplication further we have to switch to another architecture.

The Type III architecture has speedup factors ranging from 99.3 to 163.8 for the one multiplier version and factors ranging from 135.1 to 187.8 for the two multiplier version. This situation is different from the Type II design, where even the slowest 2 multiplier version is faster than the fastest 1 multiplier version.

The hardware cost for the 2 multiplier version is in terms of slices used about twice as high as the hardware cost of the 1 multiplier version using the same digitsize. We can see from Table 7.5 that the two multiplier version of digitsize D is always a little bit faster than the version with one multiplier of twice the digitsize 2D, though they use about the same number of slices. For example Type III with 2 multipliers and $D = 16$ uses 1678 slices and yields a speedup factor of 173.0, the version with 1 multiplier and $D = 32$

Design Option	Digit-size	Divisor Double [μs]	Divisor Add [μs]
Type I: SW only		6290	7630
Type II: 1 Multiplier in HW	4	387	467
	8	374	453
	16	368	446
	32	362	439
Type II: 2 Multiplier in HW	4	354	424
	8	350	419
	16	348	418
	32	346	416
Type III: AU in HW (1 Multiplier)	4	61	76
	8	46	58
	16	41	52
	32	36	46
Type III: AU in HW (2 Multiplier)	4	45	55
	8	38	47
	16	34	43
	32	32	39
Type IV: HW only	4	49	59
	8	36	44
	16	30	37
	32	26	33

Table 7.7: HECC Timings Group Operations

uses 1505 slices and yields a speed up of 163.8. Another interesting finding is that the Type IV design with a digitsize of $D = 4$ is actually slower than the Type III design with two multipliers and the same digitsize. We have to keep in mind that this extra speed comes at the cost of more slices used by the Type III design though. For $D = 8$ the Type IV design is already faster than the Type III design, because the performance of Type IV scales much better with the digitsize. This circumstance motivates to also implement a Type IV design with two multipliers in hardware to see how it performs, but due to time constraints, we did not implement this option. This task remains for future work. The speedup factors for the Type IV design range from 131.6 for $D = 4$ to

Design Option	Digit-size	gf2mMul [clock cycles]	gf2mSquare [clock cycles]	gf2mAdd [clock cycles]
Type I: Sw only		4850	511	76
Type II: Multiplier in HW	4	21+2+150=173	173	76
	8	11+2+150=163	163	76
	16	6+2+150=158	158	76
	32	3+2+150=155	155	76
Type III: AU in HW	4	21+6=27	4	6
	8	11+6=17	4	6
	16	6+6=12	4	6
	32	3+6=9	4	6
Type IV: HW only	4	21+6=27	4	6
	8	11+6=17	4	6
	16	6+6=12	4	6
	32	3+6=9	4	6

Table 7.8: HECC Timings Field Arithmetic

248.4 for $D = 32$.

The maximum frequencies in table 7.6 show the same pattern for design Types II and III. The maximum frequency for these implementations is only influenced by the digitsize of the multiplier, because the multiplier is the slowest component of these designs. The maximum frequency of Type IV is lower because this design has a more complex state machine compared with Type III and it uses additional RAM for the program memory. Before we move on and start comparing the ECC with the HECC implementations let us draw some conclusions for HECC. As we have shown it is advantageous for HECC to execute field multiplications in parallel. For the Type II design option we found out that the best performance cost tradeoff can be achieved by using two multipliers with a low digitsize in hardware, e.g. the 2 multiplier version of Type II with a digitsize of $D = 4$. This yields better results than a single multiplier in hardware with a high digitsize. For design Type III the 2 multiplier versions are also favorably, i.e. it makes more sense to invest additional hardware in a second multiplier than to invest extra hardware in a

	Group Operation	$\mathbb{F}_{2^m}$ Add	$\mathbb{F}_{2^m}$ Square	$\mathbb{F}_{2^m}$ Mutliplication
ECC	Double	4	5	5
	Add	8	4	10
HECC	Double	21	6	31
	Add	30	5	38

Table 7.9: Number of field operations for ECC and HECC group operations

faster multiplier (with a higher digitsize). From the results we obtained so far we can say that a Type IV design with 2 multipliers in hardware would probably also yield a better cost performance tradeoff than the 1 multiplier version.

7.4 Comparison of the results for ECC and HECC

The ECC and HECC implementations we implemented have about the same level of security. Hence we chose $m = 167$ for ECC and $m = 81$ for HECC. This reduction of m when switching from ECC to HECC has a significant impact on the performance of the field arithmetic functions. For example the HECC Type I implementations of `gf2mMult` and `gf2mAdd` need less than half of the clock cycles of their ECC counterparts (see 2^{nd} row of Tables 7.4 and 7.8). The performance increase of the Type I HECC field arithmetic functions compared to the ECC ones is also true for the Type II implementations. The Type II profits especially from the fact that for HECC the amount of data, that has to be transferred via the APB bottleneck between the ARM and the FPGA, is just half of the amount of data needed for ECC. But this advantage of HECC is only true when looking at the field arithmetic functions. HECC uses in its group operations more than 5 times the amount field multiplications that ECC uses. Hence the advantage of HECC's faster field arithmetic is used up in the group operations. A result of this is that the HECC scalar multiplication in our implementation is about a factor 2 slower

than the ECC scalar multiplication on our target platform, at least when comparing implementations of the same design option that use about the same number of slices on the FPGA. But it is also due to these many more multiplications that HECC profits more from the implementation of a field multiplier in hardware as done in Type II. For this kind of implementation HECC reaches speedup factors from 16.2 to 18.1 compared to ECC with 13.7 to 14.7 . Also for HECC it is possible to do a lot of the multiplications in parallel.

When comparing HECC and ECC implementations of the same Type and digitsize then it sticks out that the cost of hardware for HECC is always just about half of the cost of the ECC implementation (see Tables 7.1, 7.5 and 7.2, 7.6). So when we compare designs with about the same hardware cost, e.g. Type III ECC with $D = 4$ with 1053 slices and Type IV HECC with $D = 8$ with 1028 slices, we see that the HECC implementation is just a little bit slower. The HECC implementation needs 8.6 ms compared to 5.3 ms for the ECC scalar multiplication.

From Table 7.9 we can see why the HECC implementations for Type III and IV do not gain as much in performance as the corresponding ECC implementations do. The reason is that HECC uses 51 $\mathbb{F}_{2^m}$ additions in its two group functions, whereas ECC just uses 14 $\mathbb{F}_{2^m}$ additions. The problem is that the field additions of ECC and HECC in hardware take the same amount of time for the Type III and IV designs, because their execution time is only influenced by the access time to the memory (see Tables 7.4 and 7.8), so for both designs an addition takes 6 clock cycles for Type III.

When we look at the maximum frequencies of HECC and ECC we see that the HECC implementations can be clocked faster. This higher maximum frequency has its reason in the lower hardware complexity of the HECC implementations.

Another important difference between the ECC and HECC implementations is the

amount of RAM they use in design options Type III and IV. For the ECC scalar multiplication we need $14 * 167Bits = 2338Bits$, whereas for HECC we only need $23 * 81Bits = 1863Bits$. This is a big advantage for HECC, because the hardware cost of RAM is quite high.

A Bibliography

[1] G. B. Agnew, R. C. Mullin, and S. A. Vanstone. An implementation of elliptic curve cryptosystems over $F_{2^{155}}$. *IEEE Journal on Selected areas in Communications*, 11(5):804–813, June 1993.

[2] ARM. AMBA Specification Rev 2.0, 1999. `http://www.arm.com/products/solutions/AMBA_Spec.html`.

[3] M. Aydos, T. Yanik, and Ç. K. Koç. High-speed Implementation of an ECC-Dased Wireless Authentication Protocol on an ARM Microprocessor. In *The 16th Annual Computer Security Applications Conference*, pages 401–409. IEEE Computer Society Press, 2000.

[4] I. Blake, G. Seroussi, and N. Smart. *Elliptic Curves in Cryptography*. Cambridge University Press, London Mathematical Society Lecture Notes Series 265, 1999.

[5] N. Boston, T. Clancy, Y. Liow, and J. Webster. Genus Two Hyperelliptic Curve Coprocessor. In B. S. Kaliski, Ç. K. Koç, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems — CHES 2002*, LNCS 2523, pages 529–539. Springer-Verlag, 2002. Updated version available at `http://www.cs.umd.edu/~clancy/docs/hec-ches2002.pdf`.

[6] M. Brown, D. Hankerson, J. López, and A. Menezes. Software Implementation of the NIST Elliptic Curves Over Prime Fields. In D. Naccache, editor, *Topics in Cryptology — CT-RSA 2001*, LNCS 2020, pages 250 – 265, Berlin, April 2001. Springer-Verlag.

[7] D.G. Cantor. Computing in Jacobian of a Hyperelliptic Curve. In *Mathematics of Computation*, volume 48(177), pages 95 – 101, January 1987.

[8] Jae Wook Chung, Sang Gyoo Sim, and Pil Joong Lee. Fast Implementation of Elliptic Curve Defined over $GF(p^m)$ on CalmRISC with MAC2424 Coprocessor. In Çetin K. Koç and Christof Paar, editors, *Workshop on Cryptographic Hardware and Embedded Systems — CHES 2000*, LNCS 1965, pages 57–70, Berlin, 2000. Springer-Verlag.

[9] T. Clancy. Analysis of FPGA-based Hyperelliptic Curve Cryptosystems. Master's thesis, University of Illinois Urbana-Champaign, December 2002.

[10] T. Clancy. FPGA-based Hyperelliptic Curve Cryptosystems. invited paper presented at AMS Central Section Meeting, April 2003.

[11] A. Menezes D. Hankerson and S. Vanstone. *Guide to Elliptic Curve Cryptography.* Springer-Verlag, Germany, first edition edition, 2004.

[12] E. DeWin, S. Mister, B. Preneel, and M. Wiener. On the Performance of Signature Schemes Based on Elliptic Curves. In J. P. Buhler, editor, *Algorithmic Number Theory: Third International Symposium (ANTS 3)*, LNCS 1423, pages 252–266. Springer-Verlag, June 21–25 1998.

[13] T. ElGamal. A public-key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, IT-31(4):469–472, 1985.

[14] G. Elias, A. Miri, and T. H. Yeap. High Performance Hyperelliptic Curve Cryptosystem on an FPGA. Personal correspondence, January 2004.

[15] A. Enge. Computing Discrete Logarithms in High-Genus Hyperelliptic Jacobians in Provably Subexponential Time. `http://www.math.waterloo.ca/Cond0_Dept/CORR/corr99.html`, 1999. Preprint.

[16] A. Enge. The Extended Euclidean Algorithm on Polynomials, and the Computational Efficiency of Hyperelliptic Cryptosystems, November 1999. Preprint.

[17] M. Ernst, S. Klupsch, O. Hauck, and S. Huss. Rapid Prototyping for Hardware Accelerated Elliptic Curve Public-Key Cryptosystems. 12th IEEE Workshop on Rapid System Prototyping (RSP01), 2001.

[18] W. Fulton. *Algebraic Curves - An Introduction to Algebraic Geometry.* W. A. Benjamin, Inc., Reading, Massachusetts, 1969.

[19] L. Gao, S. Shrivastava, and G. Sobelman. Elliptic Curve Scalar Multiplier Design Using FPGAs. In Ç. Koç and C. Paar, editors, *Workshop on Cryptographic Hardware and Embedded Systems — CHES 19999*, LNCS 1717, Berlin, August 1999. Springer-Verlag.

[20] P. Gaudry and R. Harley. Counting Points on Hyperelliptic Curves over Finite Fields. In W. Bosma, editor, *ANTS IV*, LNCS 1838, pages 297 – 312, Berlin, 2000. Springer Verlag.

[21] J. Guajardo, R. Bluemel, U. Krieger, and C. Paar. Efficient Implementation of Elliptic Curve Cryptosystems on the TI MSP430x33x Family of Microcontrollers. In K. Kim, editor, *Fourth International Workshop on Practice and Theory in Public Key Cryptography - PKC 2001*, LNCS 1992, pages 365–382, Berlin, February 13-15 2001. Springer-Verlag.

[22] N. Gura, S. Chang, H. Eberle, G. Sumit, V. Gupta, D. Finchelstein, E. Goupy, and D. Stebila. An End-to-End Systems Approach to Elliptic Curve Cryptography. In Ç. K. Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems — CHES 2001*, LNCS 1965, pages 351–366. Springer-Verlag, 2001.

[23] D. Hankerson, J. López Hernandez, and A. Menezes. Software Implementation of Elliptic Curve Cryptography Over Binary Fields. In Ç. Koç and C. Paar, editors, *Second International Workshop on Cryptographic Hardware and Embedded Systems — CHES 2000*, LNCS 1965, Berlin, 2000. Springer-Verlag.

[24] T. Hasegawa, J. Nakajima, and M. Matsui. A Practical Implementation of Elliptic Curve Cryptosystems over $GF(p)$ on a 16-bit Microcomputer. In Hideki Imai and Yuliang Zheng, editors, *First International Workshop on Practice and Theory in Public Key Cryptography — PKC'98*, LNCS 1431, pages 182–194, Berlin, 1998. Springer-Verlag.

[25] K. Itoh, M. Takenaka, N. Torii, S. Temma, and Y. Kurihara. Fast Implemenation of Public-Key Cryptography on a DSP TMS320C6201. In Çetin K. Koç and Christof Paar, editors, *Proceedings of the First Workshop on Cryptographic Hardware and Embedded Systems — CHES'99*, LNCS 1717, pages 61–72, Berlin, Germany, August 1999. Springer-Verlag.

[26] D. Johnson, A. Menezes, and S. Vanstone. The Elliptic Curve Digital Signature Algorithm (ECDSA). A Certicom Whitepaper, 2001. http://www.certicom.com/resources/w_papers/w_papers.html.

[27] H. Kim, T. Wollinger, and C. Paar. Hyperelliptic Curve Coprocessors on FPGAs: Varying from High Performance to Low Area. Preprint, April 2004.

[28] N. Koblitz. Elliptic Curve Cryptosystems. *Mathematics of Computation*, 48:203–209, 1987.

[29] N. Koblitz. A Family of Jacobians Suitable for Discrete Log Cryptosystems. In Shafi Goldwasser, editor, *Advances in Cryptology - CRYPTO '88*, LNCS 403, pages 94 – 99, Berlin, 1988. Springer-Verlag.

[30] N. Koblitz. Hyperelliptic Cryptosystems. *Journal of Cryptology*, 1(3):129–150, 1989.

[31] N. Koblitz. *Algebraic Aspects of Cryptography*. Springer-Verlag, Berlin, Germany, first edition, 1998.

[32] E. Konstantinou, Y. Stamatiou, and C. Zaroliagis. A Software Library for Elliptic Curve Cryptography. In *10th European Symposium on Algorithms - ESA 2002*, pages 625–637, Berlin, Germany, 2002. Springer-Verlag. LNCS 2461.

[33] U. Krieger. signature.c. Master's thesis, Mathematik und Informatik, Universität Essen, Fachbereich 6, Essen, Germany, February 1997.

[34] J. Kuroki, M. Gonda, K. Matsuo, J. Chao, and S. Tsujii. Fast Genus Three Hyperelliptic Curve Cryptosystems. In *The 2002 Symposium on Cryptography and Information Security, Japan — SCIS 2002*, January 2002.

[35] T. Lange. Efficient Arithmetic on Genus 2 Hyperelliptic Curves over Finite Fields via Explicit Formulae. Cryptology ePrint Archive, Report 2002/121, 2002. `http://eprint.iacr.org/`.

[36] T. Lange. Inversion-Free Arithmetic on Genus 2 Hyperelliptic Curves. Cryptology ePrint Archive, Report 2002/147, 2002. `http://eprint.iacr.org`.

[37] T. Lange. Formulae for Arithmetic on Genus 2 Hyperelliptic Curves, September 2003. `http://www.ruhr-uni-bochum.de/itsc/tanja/preprints/expl_sub.pdf`.

[38] K.H. Leung, K.W. Ma, W.K. Wong, and P.H.W. Leong. FPGA Implementation of a Microcoded Elliptic Curve Cryptographic Processor. Hong Kong, April 2000. `http://www.cse.cuhk.edu.hk/~phwl/papers/ecc_fccm00.pdf`

[39] J. López and R. Dahab. Fast Multiplication on Elliptic Curves over $GF(2^n)$. In Jr. Ç. K. Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems — CHES 1999*, LNCS 1717, pages 316 – 327. Springer-Verlag, 1999.

[40] J. López and R. Dahab. High-Speed Software Multiplication in F_{2^m}. In B. Roy and E. Okamoto, editors, *Progress in Cryptology — Indocrypt 2000*, LNCS 1977, pages 203 – 212, Berlin, 2000. Springer-Verlag.

[41] Julio López and Ricardo Dahab. An overview of elliptic curve cryptography. Available from http://citeseer.nj.nec.com/333066.html.

[42] A. Menezes, Y. Wu, and R. Zuccherato. *An Elementary Introduction to Hyperelliptic Curves*. Springer-Verlag, Berlin, Germany, first edition, 1998. In: N. Koblitz, Algebraic Aspects of Cryptography.

[43] V. Miller. Uses of Elliptic Curves in Cryptography. In H. C. Williams, editor, *Advances in Cryptology — CRYPTO '85*, LNCS 218, pages 417–426, Berlin, Germany, 1986. Springer-Verlag.

[44] Y. Miyamoto, H. Doi, K. Matsuo, J. Chao, and S. Tsuji. A Fast Addition Algorithm of Genus Two Hyperelliptic Curve. In *The 2002 Symposium on Cryptography and Information Security — SCIS 2002, IEICE Japan*, pages 497 – 502, 2002. in Japanese.

[45] NEC. *Preliminary Users Manual SoCLite*, August 2001. Document No. A15402EE1V0UM00 `http://www.ee.nec.de/_pdf/A15402EE1V0UM00.PDF`.

[46] K. Nguyen. Curve based cryptography - the state of the art in smart card environments. In *6rd Workshop on Elliptic Curve Cryptosystems (ECC '02)*, Essen, Germany, September 23–25 2002. Invited Contribution.

[47] S. Okada, N. Torii, K. Itoh, and M. Takenaka. Implementation of Elliptic Curve Cryptographic Coprocessor over $GF(2^m)$ on an FPGA. In Çetin K. Koç and

Christof Paar, editors, *Proceedings of the Second Workshop on Cryptographic Hardware and Embedded Systems — CHES 2000*, pages 25–52, Berlin, Germany, 2000. Springer-Verlag.

[48] G. Orlando. *Efficient Elliptic Curve Processor Architectures for Field Programmable Logic*. PhD thesis, Dept. of ECE, Worcester Polytechnic Institute, March 2002.

[49] G. Orlando and C. Paar. A High-Performance Reconfigurable Elliptic Curve Processor for $GF(2^m)$. In Ç. K. Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems — CHES 2000*, LNCS 1965, pages 41 –56. Springer-Verlag, 2000.

[50] G. Orlando and C. Paar. A Scalable $GF(p)$ Elliptic Curve Processor Architecture for Programmable Hardware. In Ç. K. Koç, D. Naccache, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems — CHES 2001*, volume LNCS 2162, pages 348–363. Springer-Verlag, 2001.

[51] J. Pelzl. Hyperelliptic Cryptosystems on Embedded Microprocessor. Master's thesis, Department of Electrical Engineering and Information Sciences, Ruhr-Universitaet Bochum, Bochum, Germany, September 2002.

[52] J. Pelzl, T. Wollinger, J. Guajardo, and C. Paar. Hyperelliptic Curve Cryptosystems: Closing the Performance Gap to Elliptic Curves. In C. D. Walter, Ç. K. Koç, and C. Paar, editors, *Workshop on Cryptographic Hardware and Embedded Systems — CHES 2003*, LNCS 2779, pages 349 – 365. Springer-Verlag, September 2003.

[53] J. Pelzl, T. Wollinger, and C. Paar. Low Cost Security: Explicit Formulae for Genus-4 Hyperelliptic Curves. In M. Matsui and R. Zuccherato, editors, *Tenth Annual Workshop on Selected Areas in Cryptography — SAC 2003*, LNCS 3006. Springer-Verlag, 2003.

[54] J. Pelzl, T. Wollinger, and C Paar. *Embedded Cryptographic Hardware: Design and Security*, chapter Special Hyperelliptic Curve Cryptosystems of Genus Two: Efficient Arithmetic and Fast Implementation. Nova Science Publishers, NY, USA, 2004. editor Nadia Nedjah.

[55] J. Pelzl, T. Wollinger, and C. Paar. High Performance Arithmetic for Special Hyperelliptic Curve Cryptosystems of Genus Two. In *International Conference on*

Information Technology: Coding and Computing - ITCC 2004. IEEE Computer Society, April 2004.

[56] I. Riedel. Security in Ad-hoc Networks: Protocols and Elliptic Curve Cryptography on an Embedded Platform. Master's thesis, Department of Electrical Engineering and Information Sciences, Ruhr-Universitaet Bochum, Bochum, Germany, April 2003.

[57] M. Rosner. Elliptic curve cryptosystems on reconfigurable hardware. Master's thesis, ECE Department, Worcester Polytechnic Institute, Worcester, Massachusetts, USA, May 1998.

[58] Y. Sakai and K. Sakurai. Design of Hyperelliptic Cryptosystems in small Characteristic and a Software Implementation over $\mathbb{F}_{2^n}$. In K. Ohta and D. Pei, editors, *Advances in Cryptology - ASIACRYPT '98*, LNCS 1514, pages 80 – 94, Berlin, 1998. Springer Verlag.

[59] Y. Sakai and K. Sakurai. On the Practical Performance of Hyperelliptic Curve Cryptosystems in Software Implementation. In *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, volume E83-A NO.4, pages 692 – 703, April 2000.

[60] Y. Sakai, K. Sakurai, and H. Ishizuka. Secure Hyperelliptic Cryptosystems and their Performance. In H. Imai and Y. Zheng, editors, *Public Key Cryptography: First International Workshop on Practice and Theory in Public Key Cryptography — PKC'98*, LNCS 1431, pages 164 – 181, Berlin, 1998. Springer-Verlag.

[61] R. Schroeppel, H. Orman, S. O'Malley, and O. Spatscheck. Fast key exchange with elliptic curve systems. In D. Coppersmith, editor, *Advances in Cryptology — CRYPTO '95*, LNCS 963, pages 43–56, Berlin, Germany, 1995. Springer-Verlag.

[62] V. Shoup. NTL: A Libary for Doing Number Theory (version 5.3), 2003. `http://www.shoup.net/ntl/index.html`.

[63] J. H. Silverman. *The Arithmetic of Elliptic Curves*. Springer-Verlag, New York, USA, 1986.

[64] J. H. Silverman and J. Tate. *Rational Points on Elliptic Curves*. Springer-Verlag, 1992.

[65] N. Smart. On the Performance of Hyperelliptic Cryptosystems. In J. Stern, editor, *Advances in Cryptology – EUROCRYPT '99*, LNCS 1592, pages 165–175. Springer-Verlag, 1999.

[66] L. Song and K. K. Parhi. Low-Energy Digit-Serial/Parallel Finite Field Multipliers. *Journal of VLSI Signal Processing Systems*, 2(22):1–17, 1997.

[67] L. Song and K.K. Parhi. Efficient Finite Fields Serial/Parallel Multiplication. In *Int. Conf. Application Specific System Architectures and Processors*, pages 72–82, August 1996.

[68] S. Sutikno, R. Effendi, and A. Surya. Design and Implementation of Arithmetic Processor $F_{2^{155}}$ for elliptic curve cryptosystems. In *The 1998 IEEE Asia-Pacific Conference on Circuits and Systems*, pages 647–650, November 1998.

[69] A. Weimerskirch, C. Paar, and S. Chang Shantz. Elliptic Curve Cryptography on a Palm OS Device. In V. Varadharajan and Y. Mu, editors, *The 6th Australasian Conference on Information Security and Privacy — ACISP 2001*, LNCS 2119, pages 502–513, Berlin, 2001. Springer-Verlag.

[70] T. Wollinger. Computer Architectures for Cryptosystems Based on Hyperelliptic Curves. Master's thesis, ECE Department, Worcester Polytechnic Institute, Worcester, Massachusetts, USA, May 2001.

[71] T. Wollinger. *Software and Hardware Implementation of Hyperelliptic Curve Cryptosystems*. PhD thesis, Department of Electrical Engineering and Information Sciences, Ruhr-Universitaet Bochum, Bochum, Germany, July 2004.

[72] T. Wollinger and C. Paar. Hardware Architectures Proposed for Cryptosystems Based on Hyperelliptic Curves. In *Proceedings of the 9th IEEE International Conference on Electronics, Circuits and Systems - ICECS 2002*, volume III, pages 1159 – 1163, September 2002.

[73] T. Wollinger, J. Pelzl, V. Wittelsberger, C Paar, G. Saldamli, and Ç. K. Koç. Elliptic & Hyperelliptic Curves on Embedded μP. *ACM Transactions in Embed-*

ded Computing Systems (TECS), 2004. Special Issue on Embedded Systems and Security, to appear.

[74] A. Woodbury, D. V. Bailey, and C. Paar. Elliptic curve cryptography on smart cards without coprocessors. In *IFIP CARDIS 2000, Fourth Smart Card Research and Advanced Application Conference*, Bristol, UK, September, 2000. Kluwer.